ПРАКТИЧНА РОБОТА 04

Багатопоточне та мультипроцесне виконання коду

План

1. Багатопоточне програмування Windows-додатків (C#).
2. Попередження, уникнення та виявлення взаємоблокувань.
3. Практичні завдання.

Система оцінювання

№	Тема	К-ть балів
1.	*Захист принаймні одного завдання з роботи*	1
2.	Практичні завдання	3,6*
3.	*Здача звіту*	0,4
	Всього	**5**

* – діє бонусна система

1. ***Багатопоточне програмування Windows-додатків***

Операційні системи Linux та Windows постачають різні механізми синхронізації роботи ядра. ОС Windows має багатопоточне ядро, яке підтримує багато процесорів та додатки, що виконуються в реальному часі. Коли ядро Windows отримує доступ до глобального ресурсу на однопроцесорній системі, воно тимчасово маскує переривання (interrupts) для всіх обробників переривань, які можуть також звернутись до даного ресурсу. На мультипроцесорних системах Windows захищає доступ до глобального ресурсу, використовуючи спінлоки (spinlocks, циклічні блокування), незважаючи на те, що ядро використовує спінлоки тільки для захисту коротких сегментів коду. Крім того, для підтримки швидкодії ядро забезпечує неможливість витіснення потоку, який утримує спінлок.

Для синхронізації потоків поза ядром Windows постачає ***об'єкти диспетчера (dispatcher objects)***. За допомогою такого об'єкта потоки синхронізуються відповідно до кількох різних механізмів: м'ютексів, семафорів, подій та таймерів. Об'єкти диспетчера можуть перебувати в сигнальному (signaled) та несигнальному (nonsignaled) станах. У сигнальному стані об'єкт доступний, а потік не блокуватиметься при зверненні до нього. Об'єкт у несигнальному стані недоступний, а потік блокується при спробі доступу до цього об'єкта. Принцип роботи об'єкта диспетчера на основі м'ютекса зображено на рис. 1.

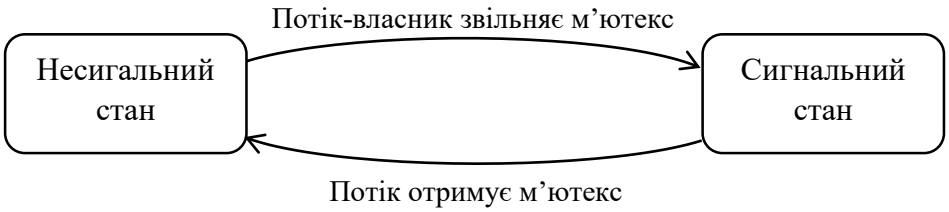


Рис. 1. Об’єкт-диспетчер на основі м’ютекса

Існує зв’язок між станом об’єкта диспетчера та станом потоку. Коли потік блокує несигнальний об’єкт диспетчера, його стан змінюється з «ready» на «waiting», а потік переміщується в чергу очікування для даного об'єкта. Коли стан об’єкта диспетчера переходить у сигнальний, ядро перевіряє, чи є потоки, які очікують на об’єкт. Якщо так, ядро переміщає один потік (може, й більше) зі стану очікування у стан готовності, в якому потік може відновити виконання. Кількість потоків, які ядро відбирає з черги очікування, залежить від типу об’єкта диспетчера, на який очікує кожен потік. Ядро відбере тільки 1 потік з черги очікування на м’ютекс, оскільки м’ютексом може «володіти» тільки один потік. Для event-об’єкта ядро відбиратиме всі потоки, які очікують на настання події.

М’ютексне блокування – ілюстрація об’єктів диспетчера та стану потоків. Якщо потік намагається отримати м’ютексний об’єкт диспетчера, який знаходиться в несигнальному стані, тоді цей потік буде зупинено та переміщено в чергу очікування для м’ютексного об’єкта. Коли м’ютекс переходить у сигнальний стан (інший потік звільняє блокування м’ютексу), очікуючий потік з голови черги перейде в стан готовності та отримає м’ютексне блокування.

Критична секція являє собою м’ютекс у режимі користувача, який часто можна отримувати та звільняти без втручання ядра. На мультипроцесорній системі критичносекційний об’єкт спочатку використовує спінлок, поки очікує, щоб інший потік звільнив об’єкт. Якщо він spins надто довго, отримуючому потоку потім буде виділено м’ютекс з ядра та передано ЦП. Критичні секції особливо ефективні, оскільки м’ютекс у ядрі виділяється тільки тоді, коли буде суперечка за даний об’єкт.

Дізнатись про інструменти для роботи з багатопоточністю та асинхронністю можна за посиланнями тут з продовженням тут.

2. Попередження, уникнення та виявлення взаємоблокувань

Взаємоблокування може виникнути тоді, коли 4 умови одночасно трапляються при багатопоточному програмуванні, використовуючи м'ютекси:

- ***Взаємне виключення.*** Принаймні один ресурс повинен підтримуватись у nonsharable режимі; тобто тільки 1 потік за раз може використовувати

ресурс. Якщо інший потік надсилає запит на цей ресурс, його робота відкладається, поки ресурс не буде звільнено.

- ***Утримування та очікування.*** Потік повинен тримати принаймні 1 ресурс та очікувати на отримання додаткових ресурсів, які на той момент утримуються іншими потоками.
- ***Відсутність витіснення.*** Ресурси не можуть витіснятись, тобто звільнити ресурс тримаючий потік може тільки добровільно після завершення цим потоком його задачі.
- ***Циклічне очікування (Circular wait)***. Множина $\{T_0, T_1, \ldots, T_n\}$ очікуючих потоків повинна існувати таким чином, щоб T_0 очікував на ресурс, який утримується T_1, T_1 – на утримуваний потоком T_2 ресурс, ..., T_{n-1} – на ресурс потоку T_n, а T_n – на ресурс потоку T_0.

2.1. Граф виділення ресурсів

Ретельніше описати взаємоблокування можна за допомогою орієнтованого графа, який називають ***графом розподілу ресурсів (system resource-allocation graph, RAG)***. Він включає множину вершин V, розбиту на 2 різних типи вузлів: $T = \{T_1, T_2, \ldots, T_n\}$ – множина активних потоків у системі та $R = \{R_1, R_2, \ldots, R_m\}$ – множина усіх типів ресурсів у системі.

Дуга від потоку T_i до ресурсу R_j позначається як $T_i \rightarrow R_j$ та описує запит очікуючого потоку T_i на тип ресурсу R_j. Її називають ***ребром запиту (request edge)***. Зворотна орієнтація записується як $R_j \rightarrow T_i$ та визначає, що екземпляр ресурсного типу R_j було виділено потоку T_i. Таку дугу називають ***ребром присвоєння (assignment edge)***.

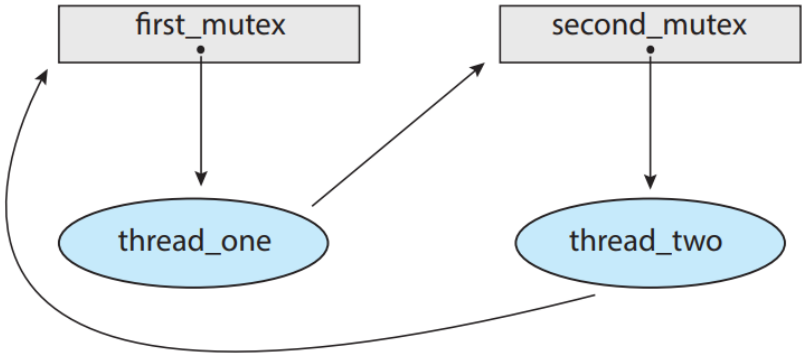
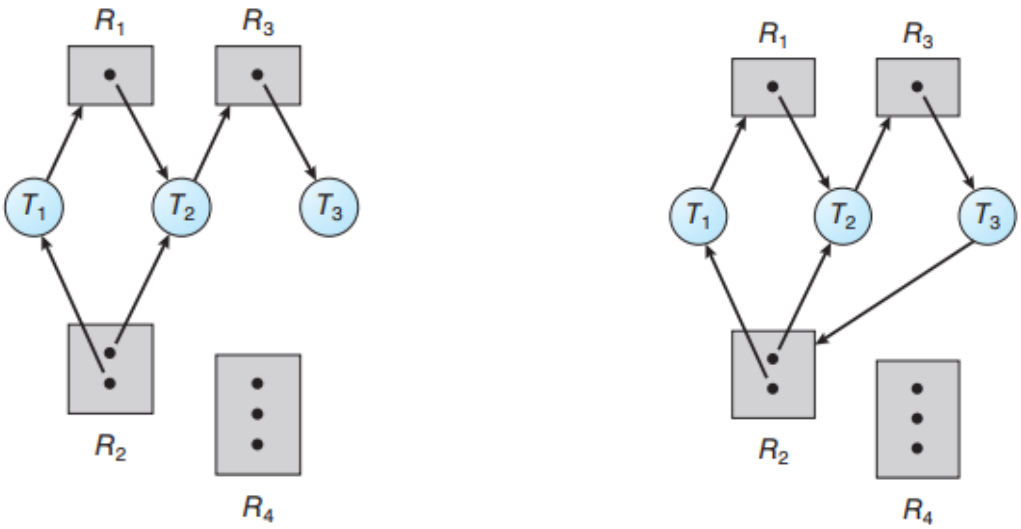


Рис. 2. RAG-граф для програми з лістингу 1

Оскільки тип ресурсу R_j може мати більше одного екземпляру, кожен з них позначаємо точкою в прямокутнику на рис. 3. Зауважте, що ребро запиту вказує тільки на прямокутник R_j, у той час як ребро присвоєння повинно також позначати одну з точок у прямокутнику.



(а) (б)

Рис. 3. RAG-граф без взаємоблокування (а) та з ним (б)

Лістинг 1.

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

Коли потік T_i звертається за екземпляром ресурсного типу R_j, ребро запиту додається до RAG-графу. Коли цей запит можна виконати, ребро

запиту ***негайно*** перетворюється на ребро присвоєння. Коли потік більше не потребує доступу до ресурсу, він звільняє ресурс. У результаті ребро присвоєння видаляється.

Граф розподілу ресурсів, зображений на рис. 3, описує наступну ситуацію:

- Множини T, R, E:
  - $T = \{T_1, T_2, T_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $\mathrm{E} = \{\mathrm{T}_1 \to \mathrm{R}_1, \mathrm{T}_2 \to \mathrm{R}_3, \mathrm{R}_1 \to \mathrm{T}_2, \mathrm{R}_2 \to \mathrm{T}_2, \mathrm{R}_2 \to \mathrm{T}_1, \mathrm{R}_3 \to \mathrm{T}_3\}$
- Екземпляри ресурсів:
  - Один екземпляр ресурсу типу R_1
  - Два екземпляри ресурсу типу R_2
  - Один екземпляр ресурсу типу R_3
  - Три екземпляри ресурсу типу R_4
- Стани потоків:
  - Потік T_1 утримує екземпляр ресурсу типу R_2 та очікує на екземпляр ресурсу типу R_1.
  - Потік Т2 тримає екземпляр R_1 та екземпляр R_2, а очікує на екземпляр R_3.
  - Потік Т3 утримує екземпляр R_3.

Відповідно до означення такого графу, якщо він не містить циклів, тоді жоден з потоків не призводить до взаємоблокування. Якщо цикл присутній, дедлок теж може існувати.

Якщо тип ресурсу має рівно один екземпляр, тоді цикл передбачає, що трапилось взаємоблокування. Якщо цикл задіює тільки множину типів ресурсу, кожен з яких має лише один екземпляр, трапляється взаємоблокування. Кожен потік, включений в цикл, блокується. У такому випадку цикл у графі є необхідної і достатньою умовою для існування дедлоку.

Якщо кожен тип ресурсу має кілька екземплярів, тоді цикл не обов'язково передбачає наявність взаємоблокування. Тоді цикл у графі є необхідною, але недостатньою умовою для існування дедлоку.

Відповідно до рис. 3б існують два мінімальні цикли в системі:

$$T_1 \to R_1 \to T_2 \to R_3 \to T_3 \to R_2 \to T_1$$

$$T_2 \to R_3 \to T_3 \to R_2 \to T_2$$

Потоки T_1, T_2 та T_3 взаємоблокуються. Потік T_2 очікує на ресурс R_3, який утримується потоком T_3. Потік T_3 очікує, поки або потік T_1, або потік T_2 звільнить ресурс R_2. Крім того, потік T_1 очікує, щоб потік T_2 звільнив ресурс R_1.

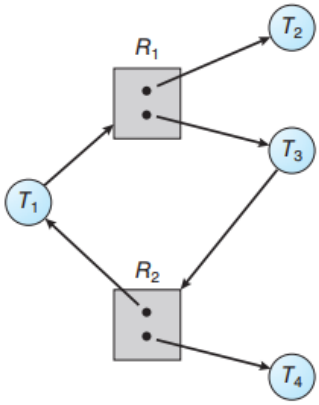


Рис. 4. RAG-граф з циклом, але без взаємоблокування

Тепер розглянемо граф розподілу ресурсів, зображений на рис. 4. У ньому теж присутній цикл

$$T_1 \to R_1 \to T_3 \to R_2 \to T_1,$$

проте відсутнє взаємоблокування. Зверніть увагу, що потік T_4 може звільняти свій екземпляр ресурсу типу R_2. Даний ресурс потім може передаватись потоку T_3, розриваючи цикл.

2.2. ***Методи обробки взаємоблокувань.***

Загалом ми можемо вирішувати проблему взаємоблокування одним з трьох способів:

- можна ігнорувати проблему та вдавати, що взаємоблокування ніколи не трапляються в системі;
- можна використовувати протокол для запобігання (prevent) та уникнення (avoid), забезпечуючи неможливість переходу системи у взаємоблокований стан;
- дозволити системі входити у взаємоблокований стан, детектувати його та відновитись з нього.

Перше вирішення використовується більшістю ОС, зокрема Linux та Windows. Розробники ядра ОС та додатків вирішують, чи писати програми для обробки взаємоблокувань, зазвичай використовуючи підходи з другого вирішення. Деякі системи – зокрема бази даних – приймають третій вихід ситуації, дозволяючи дедлокам траплятись, а потім виконуючи відновлення стану.

У даному практичному занятті розглянемо перші два вирішення з алгоритмами. Спочатку варто зазначити, що деякі дослідники вважають, що жоден з представлених виходів сам по собі не здатний вирішити весь спектр проблем виділення ресурсів в операційних системах. Базові підходи слід комбінувати, проте дозволяти обирати оптимальний для кожного класу ресурсів у системі.

Для забезпечення неможливості настання взаємоблокування система може застосувати або ***схему запобігання взаємоблокувань (deadlock-prevention scheme)***, або ***схему уникнення взаємоблокувань (deadlock-avoidance scheme)***. Запобігання дедлокам постачає набір методів, які роблять неможливими хоча б одну з чотирьох умов пункту 2. Це здійснюється шляхом обмеження здійснення запитів потоками на потрібний ресурс.

Уникнення взаємоблокувань вимагає, щоб для ОС була надана додаткова розширена інформація щодо ресурсів, якими цікавиться потік та які використовує протягом свого існування. За допомогою цієї інформації ОС може вирішити, очікувати потоку на ресурс чи ні. Для цього система повинна розглянути доступні на той момент ресурси, виділені кожному потоку ресурси та майбутні звернення та звільнення ресурсів кожним потоком.

Якщо система не задіює алгоритму запобігання або усунення взаємоблокування, воно може трапитись. У такому середовищі система може постачати алгоритм, який оглядає стан системи з метою визначення, чи трапився дедлок та алгоритм відновлення зі стану взаємоблокування.

2.3. Запобігання взаємоблокуванням

Як було зазначено в пункті 2, для запобігання взаємоблокуванню потрібно порушити виконання однієї з умов настання дедлоків:

Взаємне виключення. Умова взаємного виключення потоків повинна виконуватись. Тобто принаймні 1 ресурс повинен бути nonsharable. Спільні ресурси не вимагають ексклюзивного доступу до них, тому не можуть бути задіяними в дедлоці. Доступні тільки на зчитування файли – хороший приклад спільного ресурсу. Якщо кілька потоків намагаються відкрити такий файл одночасно, їм можуть надати одночасний (simultaneous) доступ до файлу. Потік ніколи не очікує на спільний ресурс. Проте загалом запобігти взаємоблокуванню, забороняючи взаємовиключення, не можна, оскільки деякі ресурси відразу нерозподілювані (nonsharable). Наприклад, м’ютекс-блокування не може бути одночасно спільним для кількох потоків.

Утримування та очікування. Для запобігання настанню такої умови в системі потрібно гарантувати, що потік, який подав запит на ресурс, сам не тримає інші ресурси. Один з можливих протоколів – вимога до кожного потоку про здійснення всіх запитів та виділень ресурсів до початку виконання всіх потоків. Для більшості додатків такий підхід непрактичний у зв’язку з динамічною природою звернень до ресурсів.

Альтернативний підхід дозволяє потоку звертатись до ресурсів лише тоді, коли він сам жодного ресурсу не має. Перед появою можливості

звертатись за будь-якими додатковими ресурсами, потік повинен звільнити всі ресурси, які йому в цей момент були виділені.

Обидва підходи мають 2 основних недоліки:

1) Може бути низьке використання ресурсів, оскільки вони можуть виділятись, проте не використовуватись тривалий час. Наприклад, потоку може виділитись м'ютекс на весь час його виконання, хоча він буде потрібним лише в короткому проміжку часу.
2) Можливе голодування (starvation). Потік, який потребує кількох популярних ресурсів може очікувати нескінченно довго, оскільки принаймні один з цих ресурсів завжди виділений іншому потоку.

Відсутність витіснення. Передбачає неможливість витіснення ресурсів, які вже були виділені потокам. Для забезпечення невиконання цієї умови можна використовувати такий підхід: якщо потік тримає деякі ресурси та звертається за іншим ресурсом, який неможливо йому негайно виділити, усі ресурси, які в цей момент тримає потік, витісняються (неявно звільняються). Витіснені ресурси додаються до списку ресурсів, на які очікує потік. Потік буде перезапущено тільки тоді, коли він зможе заново отримати як всі свої старі ресурси, так і нові, за якими звертався.

З іншого боку, якщо потік звертається за деякими ресурсами, спочатку перевіряємо, чи доступні вони. Якщо так – виділяємо, інакше – перевіряємо, чи виділені вони іншому потоку, який очікує на додаткові ресурси. За виконання останньої умови витісняємо бажані ресурси з очікуючого потоку та виділяємо їх потоку, який подав на них запит. Якщо ресурси й недоступні, й не тримаються очікуючим потоком, запитуючий потік повинен чекати. Протягом цього чекання деякі його ресурси можуть бути витіснені, проте лише за умови що інший потік звернувся за ними. Потік може перезапускатись лише тоді, коли йому виділили нові бажані ресурси, а в цей момент він відновлює старі ресурси, витіснені в процесі його очікування.

Цей протокол часто застосовується до ресурсів, чий стан можна просто зберегти та відновити пізніше, зокрема, регістри ЦП та транзакції баз даних. Загалом він не може бути застосованим до таких ресурсів, як м'ютекси та семафори, які й є тими типами ресурсів, для яких взаємоблокування трапляється найчастіше.

Циклічне очікування. Попередні 3 способи запобігання дедлокам загалом непрактичні в більшості ситуацій. Проте остання умова надає можливість практичного вирішення. Один із способів забезпечити невиконання цієї умови – сортувати всі ресурсні типи та вимагати, щоб кожен потік звертався за ресурсами у зростаючому порядку нумерації.

Для ілюстрації припустимо, що $R = \{R_1, R_2, \ldots, R_m\}$ – множина ресурсних типів. Присвоюємо кожному ресурсному типу унікальний цілочисельний ідентифікатор, який дозволить порівняти два ресурси та визначити їх впорядкування. Формально, визначаємо функцію $F: R \to N$, де N – множина натуральних чисел. Програмно впровадити цю схему можна шляхом впорядкування всіх синхронізаційних об'єктів у системі. Наприклад, впорядкування блокувань (lock ordering) у Pthread-програмі з лістингу 1 може бути таким:

```
F(first_mutex) = 1
F(second_mutex) = 5
```

Звідси, отримаємо наступний протокол запобігання взаємоблокуванням: кожний потік може звернутись за ресурсами тільки у зростаючому порядку їх нумерації. Тобто для ресурсів R_i та R_j обов'язкове виконання умови $F(R_j) > F(R_i)$. З іншого боку, можемо вимагати, що потік, який звертається за ресурсом R_j, повинен звільнити усі ресурси R_i, для яких $F(R_i) \geq F(R_j)$. Також зауважте, що якщо потрібні кілька екземплярів ресурсного типу, на їх усіх повинен бути єдиний запит. Якщо використовуються ці два підходи, тоді циклічне очікування не може статись.

Зверніть увагу на те, що впорядкування блокувань не гарантує запобігання дедлокам, якщо блокування можна динамічно отримувати. Наприклад, нехай буде доступна функція, яка переказує гроші між 2 рахунками. Для запобігання гонитві даних кожен рахунок має асоціюватись з м'ютексом, який отримується за допомогою функції lock(), як у лістингу 2. Дедлок можливий, якщо обидва потоки одночасно викликають функцію transaction(), націлену на різні рахунки. Один потік викличе так:

```
transaction(checking account, savings account, 25.0)
```

а інший – так:

```
transaction(savings account, checking account, 50.0)
```

2.4. Уникнення взаємоблокувань

Алгоритми запобігання взаємоблокуванням позбавляються дедлоків, обмежуючи способи здійснення звернень за ресурсами. Можливі побічні ефекти запобігання взаємоблокуванням – низький рівень використання пристрою та урізана пропускна здатність.

Альтернативний спосіб уникнення дедлоків – вимагати додаткову інформацію щодо того, як використовуються ресурси, заплановані до виділення. Наприклад, у системі з ресурсами R_1 та R_2 даній системі може стати в нагоді інформація про те, що потік P звернеться спочатку за ресурсом R_1, а потім – за R_2 перед тим, як звільнити обидва ресурси. У той же час потік Q

звернеться за ресурсом R_2, а потім – за R_1. Знаючи це, система зможе вирішити, чи повинен потік очікувати перед кожним запитом на ресурс, щоб уникнути в майбутньому можливого дедлоку. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each thread, and the future requests and releases of each thread.

Лістинг 2.

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);

   acquire(lock1);
      acquire(lock2);

         withdraw(from, amount);
         deposit(to, amount);

      release(lock2);
   release(lock1);
}
```

Різні алгоритми, які використовують цей підхід, відрізняються об'ємом і типом потрібної інформації. Найпростіша та найбільш корисна модель вимагає, щоб кожний потік оголошував максимальну кількість ресурсів кожного типу, які можуть знадобитись цьому потоку. declare the maximum number of resources of each type that it may need. Given this a priori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist. The resource-allocation state is defined by the number of available and allocated resources and the maximum demands of the threads. In the following sections, we explore two deadlock-avoidance algorithms.

Безпечний стан. Стан системи безпечний, якщо вона може виділити ресурси кожному потоку в деякому порядку та все ще уникнути взаємоблокування. Більш формально говорять, що система в безпечному стані тільки тоді, коли існує ***безпечна послідовність (safe sequence)***. Послідовність потоків $< T_1, T_2, \ldots, T_n >$ є безпечною, якщо для кожного потоку T_i можливі його звернення можуть задовольнятись доступними в цей момент ресурсами та ресурсами, які тримають усі потоки T_j, де $j < i$. In this situation, if the

resources that T_i needs are not immediately available, then T_i can wait until all T_j have finished. When they have finished, T_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When T_i terminates, T_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

Безпечний стан – невзаємоблокований, а взаємоблокований стан – небезпечний. Проте не всі небезпечні стани є дедлоками (рис. 5). Небезпечний стан може вести до взаємоблокування. Поки система в безпечному стані, вона може уникнути небезпечних (заблокованих) станів.

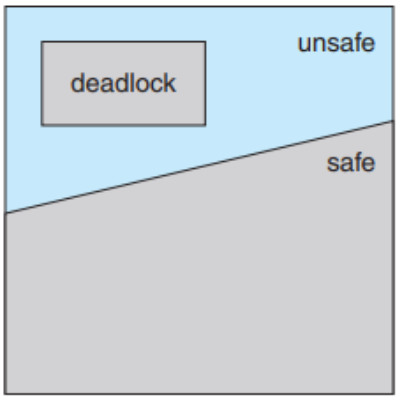


Рис. 5. Простори безпечних, небезпечних та заблокованих станів

Для ілюстрації розглянемо систему з 12 ресурсами та трьома потоками: T_0, T_1 та T_2. Потік T_0 потребує 10 ресурсів, потоку T_1 можуть знадобитись 4, а для T_2 – до 9. Припустимо, що в момент часу t_0, потік T_0 тримає 5 ресурсів, T_1 – 2 ресурси, a is holding two resources, and thread T_2 is holding two resources. (Thus, there are three free resources.)

Таблиця 1

	Максимальні потреби	**Поточні потреби**
T_0	10	5
T_1	4	2
T_2	9	2

At time t_0, the system is in a safe state. The sequence $< T_1, T_0, T_2 >$ satisfies the safety condition. Thread T_1 can immediately be allocated all its resources and then return them (the system will then have five available resources); then thread T_0 can get all its resources and return them (the system will then have ten available resources); and finally thread T_2 can get all its resources and return them (the system will then have all twelve resources available).

Система може переходити з безпечного стану в небезпечний. Припустимо, в момент часу t_1 , thread T_2 requests and is allocated one more resource. The system is no longer in a safe state. At this point, only thread T_1 can be allocated all its resources. When it returns them, the system will have only four

available resources. Since thread T_0 is allocated five resources but has a maximum of ten, it may request five more resources. If it does so, it will have to wait, because they are unavailable. Similarly, thread T_2 may request six additional resources and have to wait, resulting in a deadlock. Our mistake was in granting the request from thread T_2 for one more resource. If we had made T_2 wait until either of the other threads had finished and released its resources, then we could have avoided the deadlock.

Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock. The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state.

Whenever a thread requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or the thread must wait. The request is granted only if the allocation leaves the system in a safe state.

In this scheme, if a thread requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

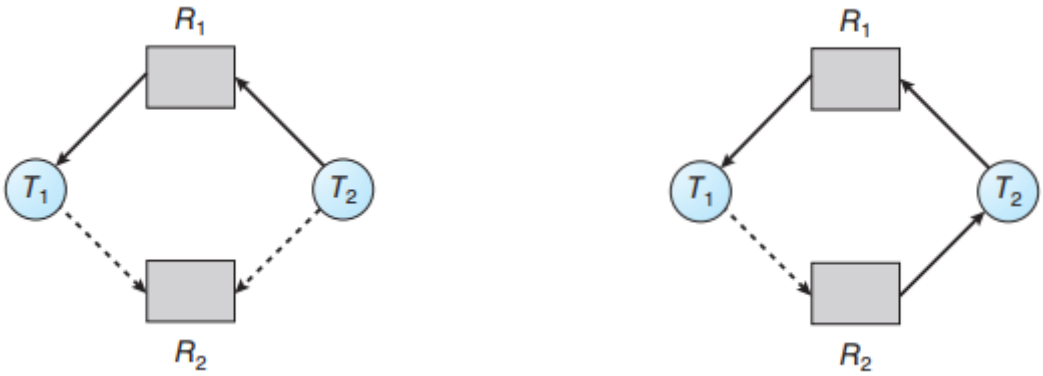
Алгоритм на базі графу розподілу ресурсів. Якщо маємо систему розподілу ресурсів з лише одним екземпляром кожного типу ресурсу, можна використати варіант графу розподілу ресурсів, визначеного для уникнення дедлоків. На додачу до вже описаних ребер запиту та присвоєння введемо новий тип дуги – ***ребро потреби (claim edge)***. Ребро потреби $T_i \rightarrow R_j$ вказує на те, що потік T_i може звертатись за ресурсом R_j у деякий момент в майбутньому. Дана дуга This edge resembles a request edge in direction but is represented in the graph by a dashed line. When thread T_i requests resource R_j, the claim edge $T_i \rightarrow R_j$ is converted to a request edge. Similarly, when a resource R_j is released by T_i, the assignment edge $R_j \rightarrow T_i$ is reconverted to a claim edge $T_i \rightarrow R_j$.

Note that the resources must be claimed a priori in the system. That is, before thread T_i starts executing, all its claim edges must already appear in the resource-allocation graph. We can relax this condition by allowing a claim edge $T_i \rightarrow R_j$ to be added to the graph only if all the edges associated with thread T_i are claim edges.

Now suppose that thread T_i requests resource R_j. The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where n is the number of threads in the system.

Якщо циклу немає, виділення ресурсу залишить систему в безпечному стані. Якщо цикл знайдено, виділення переведе систему в небезпечний стан. У такому випадку потік T_i повинен чекати на задоволення свого запиту.

Для ілюстрації алгоритму розглянемо граф розподілу ресурсів на рис. 6а. Нехай T_2 звертається за ресурсом R_2. Хоч R_2 зараз вільний, ми не можемо виділити його для T_2, оскільки така дія створить цикл у графі (рис. 6б). Якщо T_1 звертається за ресурсом R_2, а T_2 – за R_1, тоді трапиться дедлок.



(а) (б)

Рис. 6. Граф розподілу ресурсів для уникнення дедлоку

2.5. *Алгоритм банкіра*

Алгоритм на базі графу розподілу ресурсів незастосовний до систем розподілу ресурсів з кількома екземплярами кожного типу ресурсів. Описаний у подальшому алгоритм уникнення взаємоблокувань (deadlock-avoidance algorithm) може застосовуватись до таких систем, проте менш ефективний, ніж схема на базі графу розподілу ресурсів. Цей алгоритм широко відомий під назвою алгоритм банкіра. Така назва зумовлена можливим використанням алгоритму в банківській системі для забезпечення неможливості виділення доступної готівки так, щоб більше не було змоги задовольнити потреби всіх клієнтів.

Коли новий потік заходить у систему, він повинен оголосити максимальну кількість екземплярів кожного ресурсного типу, які можуть йому знадобитись. Це число може не перевищувати загальну кількість ресурсів у системі. Коли користувач звертається за набором ресурсів, система повинна визначити, чи виділення цих ресурсів залишить її в безпечному стані. Якщо так, ресурси будуть виділені, інакше – потік повинен чекати, поки деякий інший потік не звільнить достатньо ресурсів.

Кілька структур даних повинні підтримуватись для реалізації алгоритму банкіра. Вони описують стан системи розподілу ресурсів. Для n потоків у системі та m ресурсних типів маємо структури даних:

– **Available.** Вектор довжиною m, який вказує на кількість доступних ресурсів кожного типу. Якщо $\boldsymbol{Available}[j] == k$, тоді k екземплярів ресурсу типу R_j доступні.
– **Max.** Матриця розміром $n \times m$, яка визначає максимальну потребу кожного потоку. Якщо $\boldsymbol{Max}[i][j] == k$, тоді потік T_i може звернутись за максимум k екземплярів ресурсу типу R_j.
– **Allocation.** Матриця $n \times m$ визначає кількість ресурсів кожного типу для кожного потоку. Якщо $\boldsymbol{Allocation}[i][j] == k$, потоку T_i на цей момент виділено k екземплярів ресурсного типу R_j.
– **Need.** Матриця $n \times m$, яка вказує на решту потреби кожного потоку в ресурсах. Якщо $\boldsymbol{Need}[i][j] == k$, тоді потоку T_i може знадобитись ще k екземплярів ресурсу типу R_j для завершення задачі. Зауважте, що $\boldsymbol{Need}[i][j] = \boldsymbol{Max}[i][j] - \boldsymbol{Allocation}[i][j]$.

Ці структури даних змінюються з часом як за значеннями, так і за розміром. Для спрощення представлення алгоритму банкіра введемо наступну нотацію. Нехай X та Y – вектори довжиною n. Вважаємо, що $X \leq Y$ тоді, й тільки тоді, коли $X[i] \leq Y[i], \forall i = 1, 2, \ldots, n$. Наприклад, для $X = (1,7,3,2)$ та $Y = (0,3,2,1)$ $Y \leq X$. Також $Y < X$, якщо $Y \leq X$ та $Y \neq X$.

Може розглядати кожен рядок матриць $\boldsymbol{Allocation}$ і $\boldsymbol{Need}$ як вектори та звертатись до них $\boldsymbol{Allocation_i}$ та $\boldsymbol{Need_i}$. Вектор $\boldsymbol{Allocation_i}$ визначає ресурси, в даний момент виділені потоку T_i; вектор $\boldsymbol{Need_i}$ задає додаткові ресурси, за якими потік T_i ще може звернутись для завершення своєї задачі.

Алгоритм перевірки безпечності (Safety Algorithm)

Тепер представимо алгоритм перевірки того, чи знаходиться система в безпечному стані. Цей алгоритм може потребувати порядку $m \times n^2$ операцій, щоб визначити безпечність стану.

```
Крок 1. Ініціалізація.
        Work = Available
        Для i = 1, …, n, якщо Allocation[i] != 0 то finish[i] = false. Інакше
                                                    finish[i] = true.
Крок 2. Знаходимо i, таке, що:
        Finish [i] = false
        Request [i] <= Work
        Якщо такого i немає, переходимо до кроку 4.
Крок 3.
        Work = Work + Allocation [i]
        Finish [i] = true
        Перехід до кроку 2.
Крок 4. Якщо Finish[i] = false для деякого i від 1 до n, то система в заблокованому
стані.
          Більше того, якщо Finish[i] = false,
          то потік Ti – у стані дедлоку
```

Алгоритм запиту ресурсу (Resource-Request Algorithm)

Опишемо алгоритм для визначення, чи безпечно давати дозвіл на виконання запиту. Нехай $\boldsymbol{Request}_i$ – це вектор-запит (request vector) для потоку T_i. Якщо $\boldsymbol{Request}_i[j] == k$, тоді потік T_i бажає k екземплярів ресурсу типу R_j. Коли запит на ресурси здійснено потоком T_i, потрібні наступні дії:

1. If $\boldsymbol{Request}_i \leq \boldsymbol{Need}_i$, go to step 2. Otherwise, raise an error condition, since the thread has exceeded its maximum claim.
2. If $\boldsymbol{Request}_i \leq \boldsymbol{Available}$, go to step 3. Otherwise, T_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread T_i by modifying the state as follows:

$$\boldsymbol{Available} = \boldsymbol{Available} - \boldsymbol{Request}_i$$
$$\boldsymbol{Allocation}_i = \boldsymbol{Allocation}_i + \boldsymbol{Request}_i$$
$$\boldsymbol{Need}_i = \boldsymbol{Need}_i - \boldsymbol{Request}_i$$

If the resulting resource-allocation state is safe, the transaction is completed, and thread T_i is allocated its resources. However, if the new state is unsafe, then T_i must wait for $\boldsymbol{Request}_i$, and the old resource-allocation state is restored.

Приклад.

Для ілюстрації роботи алгоритму банкіра розглянемо систему з 5 потоків, від T_0 до T_4 та трьома ресурсними типами A, B й C. Ресурсний тип A має 10 екземплярів, ресурс B – 5, а ресурс C – 7:

	Allocation	*Max*	*Available*
	A B C	A B C	A B C
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

Вміст матриці $\boldsymbol{Need}$ визначається як $\boldsymbol{Max - Allocation}$:

	Need
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

Вважаємо, що система зараз у безпечному стані (safe state). Послідовність $\langle T_1, T_3, T_4, T_2, T_0 \rangle$ задовольняє критерій безпечності. Нехай потік

T_1 звертається за ресурсом типу A та двома екземплярами ресурсу типу C, тобто $\boldsymbol{Request_1} = (1,0,2)$. Для визначення можливості негайного дозволу на виконання спочатку перевіряємо, що $\boldsymbol{Request_1} \leq \boldsymbol{Available}$, тобто $(1,0,2) \leq (3,3,2)$, що є так. Потім вдамо, що цей запит було здійснено, що призведе до переходу в новий стан:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Необхідно визначити, чи такий новий системний стан безпечний. Виконаємо алгоритм перевірки безпечності, у результаті якого знайдемо послідовність $\langle T_1, T_3, T_4, T_0, T_2 \rangle$, яка відповідає вимогам безпеки. Таким чином, можемо негайно дозволити виконання запиту потоку T_1.

Проте зауважте, що коли система в цьому стані, запит на (3,3,0) від потоку T_4 не дозволяється, оскільки ресурси недоступні. Крім того, запит на (0,2,0) від T_0 теж недозволений, хоч навіть ресурси доступні, оскільки результуючий стан не буде безпечним.

2.6. Виявлення (detection) взаємоблокувань

Якщо система не задіює алгоритми запобігання чи уникнення взаємоблокувань, може трапитись дедлок. У такому середовищі система може забезпечити:

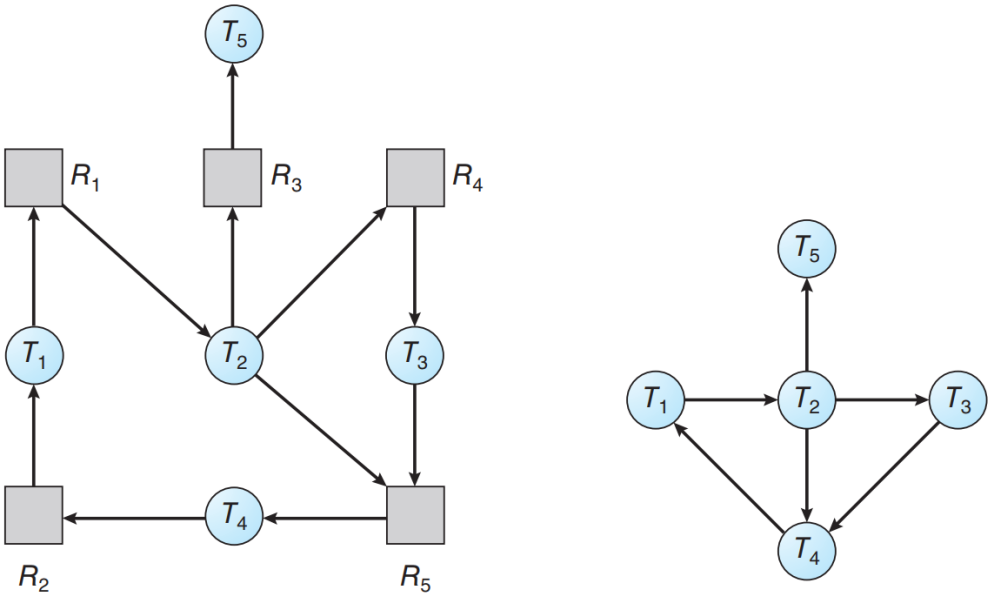
- алгоритм, який перевірить стан системи на наявність взаємоблокування;
- алгоритм для відновлення зі стану взаємоблокування.

На даному етапі слід зауважити, що схема «виявлення та відновлення» потребує накладних витрат, які включають не лише runtime-затрати на підтримку необхідної інформації та виконання алгоритму виявлення, а й потенційні втрати, притаманні відновленню із заблокованого стану.

Один екземпляр кожного ресурсного типу

Якщо всі ресурси мають тільки 1 екземпляр, можна визначити алгоритм виявлення дедлоку, який використовує варіант графу розподілу ресурсів, який називають ***графом очікування (wait-for graph)***. Даний граф отримується з графу розподілу ресурсів шляхом видалення вузлів ресурсів та прибирання відповідних ребер. Точніше, ребро $T_i \rightarrow T_j$ у графі очікування передбачає, що потік T_i очікує, поки потік T_j звільнить ресурс, який потрібний потоку T_i. Ребро

$T_i \rightarrow T_j$ існує в графі очікування тоді й тільки тоді, коли відповідний граф розподілу ресурсів містить 2 ребра $T_i \rightarrow R_q$ та $R_q \rightarrow T_j$ для деякого ресурсу R_q. На рис. 7 зображено граф розподілу ресурсів та відповідний граф очікування.



(а) (б)

Рис. 7. (а) Граф розподілу ресурсів. (b) Відповідний граф очікування.

Як і раніше, взаємоблокування існує в системі тільки тоді, коли граф очікування містить цикл. Для виявлення дедлоків система підтримує граф очікування та періодично викликає алгоритм, який шукає цикл у графі. Такому алгоритму потрібно $O(n^2)$ операцій, де n – кількість вершин графу.

Кілька екземплярів ресурсного типу

Схема на базі wait-for графу незастосовна до систем розподілу ресурсів з кількома екземплярами кожного ресурсного типу. Перейдемо до алгоритму виявлення (detection) взаємоблокувань для такої системи. Він задіює кілька змінних у часі структур даних, аналогічних до структур в алгоритмі банкіра:

- **Available.** Вектор довжини m вказує на кількість доступних ресурсів кожного типу.
- **Allocation.** Матриця $n \times m$, яка визначає кількість ресурсів кожного типу, у поточний момент виділених для кожного потоку.
- **Request.** Матриця $n \times m$, вказує поточний запит на ресурс для кожного потоку. Якщо $\boldsymbol{Request}[i][j] = k$, тоді потік T_i звертається ще за k екземплярами ресурсного типу R_j.

Відношення $\leq$ між двома векторами визначалось у пункті 2.5. Для спрощення запису знову розглядатимемо рядки матриць $\boldsymbol{Allocation}$ та $\boldsymbol{Request}$ як вектори $\boldsymbol{Allocation}_i$ та $\boldsymbol{Request}_i$ відповідно. Алгоритм

виявлення досліджує кожну можливу послідовність виділення ресурсів для потоків, які працюють, але ще не завершились.

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively. Initialize ***Work*** = ***Available.*** For $i = 0, 1, ..., n-1$, if $\boldsymbol{Allocation}_i \neq 0$, then ***Finish***[$i$] = ***false.*** Otherwise, ***Finish***[i] = ***true.***
2. Find an index i such that both
   a. ***Finish***[i] == ***false***
   b. $\boldsymbol{Request}_i \leq \boldsymbol{Work}$

   If no such i exists, go to step 4.
3. $\boldsymbol{Work} = \boldsymbol{Work} + \boldsymbol{Allocation}_i$
   Finish[i] = ***true***
   Go to step 2.
4. If ***Finish***[i] == ***false*** for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if ***Finish***[i] == ***false,*** then thread T_i is deadlocked.

Даний алгоритм визначає порядок $m \times n^2$ операцій, щоб визначити, чи система знаходиться у взаємоблокованому режимі.

Можливо, Вам буде цікаво, навіщо ресурси потоку T_i в кроці 3 повертаються (reclaim), як тільки визначається, що $\boldsymbol{Request}_i \leq \boldsymbol{Work}$ у кроці 2b. Ми знаємо, що T_i на цьому етапі не задіяний в дедлоці, оскільки $\boldsymbol{Request}_i \leq \boldsymbol{Work}$, тому оптимістично припускаємо, що T_i більше не вимагатиме ресурсів для завершення задачі; тобто скоро він поверне всі поточно виділені ресурси системі. Якщо припущення некоректне, взаємоблокування може трапитись пізніше. Воно буде виявлене при наступному виклику deadlock-detection алгоритму.

Для прикладу розглянемо систему з 5 потоків від T_0 до T_4 та 3 ресурсних типи A, B й C. Ресурсний тип A має 7 екземплярів, B – 2, а C – 6. Поточний стан системи такий:

	Allocation	***Request***	***Available***
	$A\ B\ C$	$A\ B\ C$	$A\ B\ C$
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

Вважаємо, що система не у взаємозаблокованому стані. Якщо виконаємо алгоритм, знайдеться послідовність $\langle T_0, T_2, T_3, T_1, T_4 \rangle$, у результаті якої $\boldsymbol{Finish}[i] == true$ для всіх i.

Тепер нехай потік T_2 здійснює 1 додатковий запит на екземпляр ресурсу C. Матриця $\boldsymbol{Request}$ змінюється так:

	Request
	$A\ B\ C$
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

Тепер вважаємо, що система заблокована. Хоч можемо повернути (reclaim) ресурси, які тримає потік T_0, кількість доступних ресурсів недостатня для виконання запитів інших потоків. Тому існує дедлок, який складається з потоків T_1, T_2, T_3 та T_4.

Коли алгоритм виявлення взаємоблокувань визначає наявність дедлоку, доступні кілька варіантів наступного кроку. Одна з можливостей – проінформувати оператора, що трапилось взаємоблокування та дати оператору вручну його вирішити. Інша можливість – автоматично відновити (recover) стан системи. Існують 2 можливості перервати циклічне очікування:

1) Просто ***перервати роботу*** одного чи кількох потоків. Варіант з перериванням роботи всіх потоків гарантує розривання дедлоку, проте обчислювально багато вартує. Якщо один із заблокованих потоків був довготривалий, всі здійснені ним обчислення потрібно скасувати та переобчислювати заново. Одиничне «вбивання» потоків до виходу з заблокованого стану призводить до значних накладних витрат, оскільки за видаленням кожного потоку/процесу доводиться перезапускати алгоритм виявлення взаємоблокувань. Порядок видалення потоків чи процесів – це політика, подібна до планування процесів. Питання зводиться до економності, щоб вийти із заблокованого стану з мінімальними витратами. Проте на вибір потоків/процесів для переривання та найменше затрати впливають багато чинників: пріоритет процесу, тривалість його роботи, абсолютна та відносно інших потоків чи процесів, кількість задіяних та потрібних ресурсних типів, кількість процесів для завершення задля переривання роботи тощо.
2) ***Витіснити деякі ресурси*** в одного чи кількох взаємоблокованих потоків. Тут потрібно вирішити 3 завдання:
   - ***Обрання жертви.*** Як і при достроковому завершенні процесу, необхідно визначити порядок витіснення з метою мінімізації витрат. Фактори можуть бути наступними: кількість утримуваних ресурсів заблокованим потоком, тривалість роботи процесу на даний момент;

- ***Відкочування (rollback) процесу.*** Якщо витіснити ресурс з процесу, що з процесом станеться? Доведеться відновити процес до деякого безпечного стану та перезапустити з нього. Такий стан на практиці знайти важко, тому здійснюють повне відкочування – перезапуск процесу заново.
- ***Голодування.*** Як гарантувати його відсутність після переривання роботи потоків/процесів? При оптимізації затрат на «вбивство» процесу може статись так, що в якості жертви завжди обиратиметься один і той же процес. Тому система має забезпечити перевірку того, що жертва обирається скінченну та невелику кількість разів. Найбільш поширений підхід – включити кількість відкочувань у cost factor.

3. Практичні завдання

Застосовуючи технологію багатопоточного програмування для Windows (C++, C#) або Linux (Pthreads), створіть програмне забезпечення, яке буде реалізувати наступні завдання:

1. ***(Потоки)*** Напишіть програму, яка створить два потоки, кожен з яких спатиме випадкову кількість секунд, а потім буде виводити повідомлення, яке включатиме ID потоку.
2. ***(Barrier)*** Припустимо, що потрібно написати програмне забезпечення для приготування страви. Наприклад, у замовленні в Mcdonalds працюватиме 3 потоки: з приготування бургеру, картоплі фрі та коли. Проте замовлення буде готове до подачі лише тоді, коли будуть готові всі його компоненти. Створіть бар'єр на 3 завдання (кожне у своєму потоці). Запустіть приготування замовлення та виведіть на екран інформацію щодо порядку запуску та приготування компонентів замовлення.
3. ***(Semaphore)*** Уявіть кабіну з 4 банкоматами, за допомогою яких відбувається обслуговування населення. Це означає, що в кожний момент часу може обслуговуватись максимум 4 людини. Створіть 6 потоків (людей у спільній черзі), кожен з яких працюватиме зі спільним семафором. Потік має виконувати 4 операції, кожна з яких сповіщатиме про її виконання та спатиме 1 секунду. Також потік повинен завершувати роботу (сповіщати про звільнення блокування (lock)).
4. ***(Condition-об'єкти)*** Уявіть, що пишете відеоплеєр, який відтворюватиме потокове відео. Тут буть задіяні мінімум два потоки: один буде буферизувати відео для подальшого відтворення, а інший – власне відтворювати відеоряд. Основна проблема синхронізації – плеєр не може відтворювати небуферизоване відео.

   Створіть програму з двома потоками. Один потік буде «буферизувати» відео – генеруватиме випадкове число від 1 до 5 (відсоток буферизації) та спатиме 1 секунду. Як тільки у буфері накопичилось ще 10% відео, можна розпочати «відтворення» (в іншому потоці, який спить 3 секунди). Виводьте повідомлення щодо всіх подій у програмі: новий відсоток при буферизації, спрацювання умови, запуск та завершення роботи потоків.
5. Диспетчери в аеропорту спрямовують літаки на посадку на різних смугах. Уявіть, що в аеропорту є 3 посадкові смуги. Нехай один літак виконує посадку протягом 7 хвилин. Якщо всі посадкові смуги зайняті, решта літаків повинні чекати, поки смуга звільниться. Змоделюйте чергу з 12 літаків, які бажають зайти на посадку (не

обов’язково всі одномоментно), а також роботу диспетчера, який спрямовуватиме їх на доступні посадкові смуги.

6. Напишіть програму, яка за допомогою алгоритму банкіра перевірятиме безпечність стану. Продемонструйте роботу алгоритму на наступній системі:

	Allocation	***Max***
	A B C D	*A B C D*
T_0	3 0 1 4	5 1 1 7
T_1	2 2 1 0	3 2 1 1
T_2	3 1 2 1	3 3 2 1
T_3	0 5 1 0	4 6 1 2
T_4	4 2 1 2	6 3 2 5

Якщо стан безпечний, програма повинна виводити порядок завершення потоків. Інакше – вказувати, що стан не є безпечним. Можете взяти для перевірки стани

Available = (0, 3, 0, 1)

Available = (1, 0, 0, 2)