9.4 УХИЛЕННЯ ВІД ВЗАЄМНИХ БЛОКУВАНЬ

Мета запобігання взаємних блокувань – забезпечити умови, що унеможливлюють виникнення тупикових ситуацій. Система повинна вміти приймати рішення про те, чи представляє виділення ресурсу небезпеку, чи ні, і виділяти його тільки в тому випадку, коли це безпечно. Розглянемо способи попередження взаємних блокувань за рахунок ретельного розподілу ресурсів.

9.4.1 Заборона запуску процесу

Розглянемо систему з n процесів і т різних типів ресурсів. З рис. 9.5 ми маємо такі вектори і матриці:

Ресурс =	$(E1, E2 \ldots Em)$	Загальна кількість кожного ресурсу в системі
Доступність =	$(A1, A2 \ldots Am)$	Загальна кількість кожного ресурсу, не виділеного процесам
Вимоги =	$\begin{pmatrix} R11, R12 \ldots R1m \\ R21, R22 \ldots R2m \\ Rn1, Rn2 \ldots Rnm \end{pmatrix}$	Запити кожного процесу на кожен з ресурсів
Розподіл =	$\begin{pmatrix} C11, C12 \ldots C1m \\ C21, C22 \ldots C2m \\ Cn1, Cn2 \ldots Cnm \end{pmatrix}$	Поточний розподіл ресурсів

Матриця вимог (запитів), в якій кожен рядок описує один з процесів, вказує максимальні вимоги кожного процесу до різних ресурсів, тобто Rij – це вимоги процесом і ресурсу j. Для забезпечення працездатності методу усунення взаємоблокувань ця інформація має бути оголошена процесом заздалегідь. Аналогічно, Cij – поточна кількість розподіленого ресурсу j, виділене процесу і.

Повинні виконуватися такі співвідношення:

$$Ei = Ai + \sum_{k=1}^{n} Cki$$

1. для всіх i: усі ресурси або вільні, або виділені.

2. Rki ≤ Ei, для всіх k і i: жоден процес не може затребувати ресурс, що перевищує його загальну кількість в системі.

3. Cki ≤ Rki для всіх k і i: жоден процес не може отримати більше ресурсів, ніж ним було затребувано.

Коли усі вказані величини визначені, ми в змозі створити стратегію усунення взаємних блокувань, яка забороняє запуск нового процесу, якщо його вимоги ресурсів можуть призвести до взаємного блокування. Новий процес Pn+1 запускається тільки якщо

$$Ei \geq R(n+1)i + \sum_{k=1}^{n} Rki \text{ для всіх } i.$$

Це означає, що запуск нового процесу станеться тільки тоді, коли можуть бути задоволені максимальні вимоги усіх поточних процесів плюс вимоги процесу, що запускається. Ця стратегія аж ніяк не є оптимальною, оскільки припускає гірше: що усі процеси пред'являть максимальні вимоги одночасно.

9.4.2 Алгоритм банкіра для одного ресурсу

Можна уникнути взаємного блокування, якщо розподіляти ресурси, дотримуючись певних правил. Серед такого роду алгоритмів найвідоміший алгоритм банкіра, який запропонував Дейкстра (Dijkstra, 1965 р., уперше реалізований в операційній системі THE в кінці 1960-х рр.), який базується на безпечних або надійних станах (safe state) [14]. **Безпечний стан** – це такий стан, для якого є, принаймні, одна послідовність подій, яка не призведе до взаємного блокування. Модель алгоритму заснована на діях банкіра, який, маючи в наявності капітал, видає кредити.

Суть алгоритму полягає в наступному. Алгоритм перевіряє, чи веде виконання кожного запиту до небезпечного стану. Якщо так, то запит відхиляється. Якщо задоволення запиту до ресурсу призводить до безпечного (надійному) стану, ресурс надається процесу. На рис. 9.9, а показані чотири клієнти: A, B, C і D, кожен з яких отримав певну кількість одиниць кредиту. Банкір знає, що не всім клієнтам миттєво знадобиться максимальна сума їх кредиту, тому для обслуговування їх потреб він зарезервував тільки 10 одиниць, а не всі 22, які потрібні клієнтам. Щоб провести аналогію з комп'ютерною системою, вважатимемо, що клієнти – це процеси, одиниці – накопичувачі на дисках, а банкір – це операційна система.

Клієнти займаються своїми справами, просячи час від часу позики (тобто, ресурси, по одному за один запит). Процесам дозволяється утримувати за собою ресурси, просячи і чекаючи виділення додаткових ресурсів. Система може або задовольнити, або відхилити кожен запит. В якийсь певний момент виникає ситуація, показана на рис. 9.9, б. Цей стан не представляє небезпеки, оскільки при двох одиницях, що залишилися, банкір може відкласти виконання будь-яких запитів, за винятком запиту клієнта C, дозволяючи C завершити свої справи і вивільнити усі чотири ресурси. Маючи у своєму розпорядженні чотири ресурси, банкір може дозволити отримати необхідні одиниці або D або B тощо.

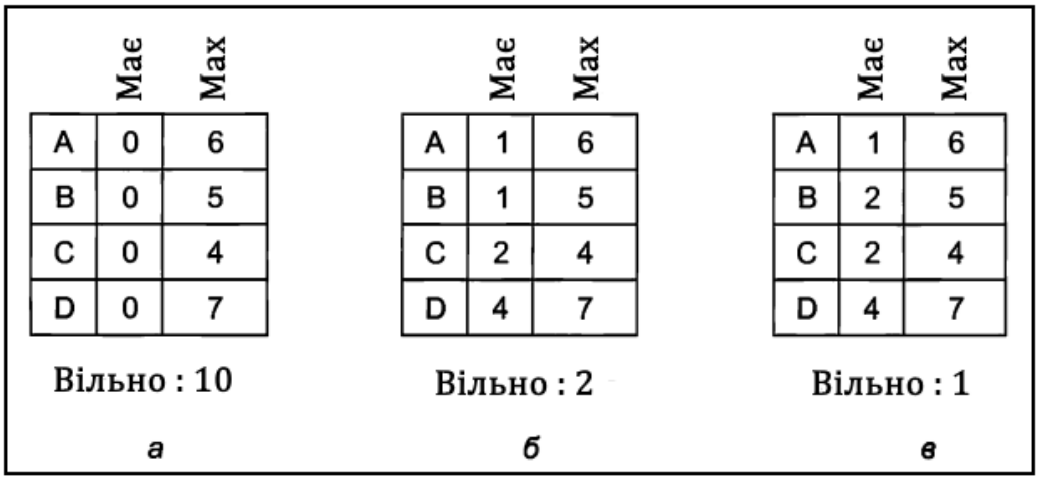

Рисунок 9.9 – Стани розподілу ресурсів: безпечне (а), безпечне (б), небезпечне (в)

Розглянемо, що вийде, якщо запит від B однієї додаткової одиниці буде задоволений в ситуації, показаній на рис. 9.9, б. Ми отримаємо небезпечну ситуацію, показану на рис. 9.9, в. Якщо усі клієнти несподівано запросять свої максимальні позики, банкір не зможе задовольнити нікого з них, і ми отримаємо взаємоблокування. Небезпечний стан не обов'язково призводить до взаємоблокування, оскільки клієнтові може не знадобитися уся доступна максимальна сума кредиту, але банкір не може розраховувати на це.

9.44.3Алгоритм банкіра для декількох типів ресурсів

Алгоритм банкіра може бути поширений на роботу з декількома ресурсами. На рис. 9.10 показано, як він працює.

Рисунок 9.10 – Алгоритм банкіра для системи з декількома типами ресурсів

На рисунку зображені дві матриці. Ліва матриця (а) показує, скільки екземплярів кожного ресурсу в даний момент виділені кожному з п'яти процесів. Права матриця (б) показує, скільки екземплярів ресурсів все ще потрібні кожному процесу для завершення його роботи. На рис. 9.6 ці матриці називалися C і R. Як і у випадку з ресурсом одного типу, процеси перед виконанням своєї роботи повинні повідомити про свої загальні потреби в ресурсах, щоб система в будь-який момент могла обчислити праву матрицю (б).

Три вектори, що зображені праворуч від матриць, показують відповідно існуючі ресурси (вектор E), зайняті ресурси (вектор P) і доступні ресурси (вектор A). Судячи зі значення вектору E, в системі є шість дисків, три плотери, чотири сканери і два приводи компакт-дисків. З них зайняті в даний момент п'ять накопичувачів на дисках, три плотери, два сканери і два приводи компакт-дисків. Цей факт можна встановити шляхом складання значень чотирьох стовпців, що відповідають ресурсам, в лівій матриці. Вектор доступних ресурсів – це різниця між кількістю присутніх в системі ресурсів, і кількістю ресурсів використовуваних нині.

Суть алгоритму:

1. Припустимо, що в системі в наявності n пристроїв, наприклад сканерів.
2. ОС приймає запит від призначеного для користувача процесу, якщо його максимальна потреба не перевищує n.

3. Користувач гарантує, що якщо ОС в змозі задовольнити його запит, то усі пристрої будуть повернені системі впродовж кінцевого часу.

4. Поточний стан системи називається надійним, якщо ОС може забезпечити усім процесам їх виконання впродовж кінцевого часу.

5. Відповідно до алгоритму банкіра виділення пристроїв можливе, тільки якщо стан системи залишається надійним.

Тепер може бути викладений алгоритм перевірки стану на безпеку:

1. Шукаємо в матриці R рядок, що відповідає процесу, чиї незадоволені потреби в ресурсах менше або дорівнюють вектору A. Якщо такого рядка не існує, то система врешті-решт увійде до стану взаємного блокування, оскільки жоден процес не зможе допрацювати до успішного завершення (передбачається, що процеси утримують усі ресурси, поки не завершать свою роботу).

2. Допускаємо, що процес, чий рядок був вибраний, просить усі необхідні йому ресурси (можливість чого гарантується) і завершує свою роботу. Відмічаємо цей процес як завершений і додаємо усі його ресурси до вектора A.

3. Повторюємо кроки 1 і 2 до тих пір, поки або усі процеси будуть помічені як завершені (у цьому випадку початковий стан може вважатися безпечним), або не залишиться процесів, чиї запити можуть бути задоволені (в цьому випадку виникне взаємне блокування).

Якщо в кроці 1 підходять для вибору декілька процесів, то неважливо, який з них буде вибраний: фонд доступних ресурсів або збільшується, або в гіршому разі залишається таким же.

Тепер повернемося до прикладу, показаному на рис. 9.10. Поточний стан безпечний. Припустимо, що процес B тепер зробив запит на сканер. Цей запит може бути задоволений, оскільки стан, що виходить в результаті, як і раніше безпечний (процес D може завершити свою роботу, потім це ж може зробити процес A або процес E, а потім і усі інші).

Уявімо тепер, що після виділення процесу B одного з двох сканерів, що залишилися, процес E зажадає останній сканер. Задоволення цього запиту зменшить значення вектору доступних ресурсів до A(1 0 0 0), що призведе до взаємоблокування. Ясно, що запит процесу E має бути на деякий час відхилений.

9.4.4 Реалізація алгоритму банкіра

Структури даних для алгоритму банкіра. Нехай в системі є n процесів і m типів ресурсів. Вектор *Available* довжини m містить інформацію про доступні ресурси. Якщо *Avaliable*[j] = k, то в системі в даний момент доступно k одиниць ресурсу j.

Матриця *Max*(n*m) відображає максимальні потреби процесів в ресурсах. Якщо *Max* [i, j] = k, то процес i може запросити не більше k одиниць ресурсу j.

Матриця Allocation(n*m) відображає фактичне виділення системою ресурсів. Якщо Allocation [i, j] = k, то процесу i в даний момент виділено системою k одиниць ресурсу j.

Матриця Need(n*m) відображає потреби процесів в ресурсах., які ще залишилися, Якщо Need [i, j] = k, то процесу i можуть знадобитися ще k одиниць ресурсу j для завершення роботи.

Має місце таке співвідношення між елементами матриць:

Need [i, j] = **Max** [i, j] – **Allocation** [i, j].

Алгоритм перевірки стану системи на безпеку. У позначеннях «Структури даних для алгоритму банкіра», розглянемо алгоритм перевірки стану системи на те, чи є він безпечним.

Введемо цілочисельний вектор *Work* (довжини m) і булевий вектор *Finish* (довжини n). Вектор *Work* відображає пробні виділення ресурсів. Вектор *Finish* представляє інформацію про завершення процесів при цьому стані системи.

Алгоритм безпеки.

Крок 1. Ініціалізація.

Work = Available

Finish [i] = false для i = 1, …, n.

Тут і надалі усі привласнення і порівняння, в яких беруть участь вектори або матриці, виконуються поелементно.

Крок 2. Знаходимо i, таке, що:

Finish [i] = false

Need [i] <= Work

Якщо такого і немає, переходимо до кроку 4.

Крок 3.

Work = Work + Allocation [i]

Finish [i] = true

Перехід до кроку 2.

Крок 4. *Якщо Finish[i] = true для усіх i = 1, ..., n,* то система в безпечному стані.

Необхідні пояснення до алгоритму:

1. Алгоритм будує безпечну послідовність номерів процесів і, якщо вона існує. На кожному кроці, після виявлення чергового елементу послідовності, алгоритм моделює звільнення і-м процесом ресурсів після його завершення.

2. На кроці 1 привласнення векторів виконується поелементно.

3. На кроці 2, *Need* – матриця потреб (n * m); Need[i] – рядок матриці, що представляє вектор потреб (довжини m) процесу і. Порівняння виконується поелементно, і його результат вважається істинним, якщо співвідношення виконане для всіх елементів векторів. Умова, що перевіряється, означає, що потреби процесу і зі знайденим номером можуть бути задоволені негайно, і процес може отримати їх і завершитися.

4. На кроці 3, Allocation [i] – рядок матриці *Allocation*, що означає поточні ресурси, виділені процесу і. За допомогою вектора *Work* моделюється звільнення ресурсів і-м процесом, після чого процесу привласнюється ознака завершенності.

Алгоритм запиту ресурсів для процесу Pi – основна частина алгоритму банкіра. Для основного алгоритму введемо вектор *Requesti* (довжини m) – вектор запитів для процесу Pi. Якщо Requesti [j] = k, то процес Pi просить k екземплярів ресурсу Rj.

Крок 1. Якщо Requesti <= Need[i], перейти до кроку 2.

Інакше – згенерувати виняткову ситуацію (процес перевищив свої максимальні потреби).

Крок 2. Якщо Requesti <= Available, перейти до кроку 3.

Інакше процес повинен чекати, оскільки ресурс недоступний.

Крок 3. Спланувати виділення ресурсу процесу Pi, модифікуючи стан системи таким чином:

Available = Available - Requesti

Allocation = Allocation + Requesti

Need [i] = Need [i] - Requesti

Викликати алгоритм перевірки безпеки отриманого стану. Якщо стан безпечний, виділити ресурс процесу Pi. Вихід.

Якщо стан небезпечний, відновити попередній стан; процес повинен чекати.

Приклад використання алгоритму банкіра. Нехай є 5 процесів – P0, …, P4, і 3 типи ресурсів – ресурс A (10 екземплярів), ресурс B (5 екземплярів) і ресурс C (7 екземплярів). Нехай стан системи в момент T0 такий:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Обчислимо матрицю потреб *Need = Max - Allocation:*

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Неважко бачити, що система знаходиться в безпечному стані. Послідовність процесів <P1, P3, P4, P2, P0> задовольняє критерію безпеки.

У продовження прикладу припустимо, що процес P1 зробив запит (1 0 2). Перевіряємо, що *Request <= Available:* <(1 0 2) <= (3 3 2) = *true.*

Задовольняємо запит. Стан системи набирає вигляду:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	1	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Виконання алгоритму безпеки показує, що послідовність процесів <P1, P3, P4, P0, P2> задовольняє критерію безпеки.

9.4.5 Недоліки алгоритму банкіра

Хоча алгоритм чудовий у теорії, на практиці він, по суті, марний, оскільки нечасто можна визначити заздалегідь, які будуть максимальні потреби процесів в ресурсах. Крім того, кількість процесів не фіксована, вона динамічно змінюється в міру входу користувачів в систему і виходу їх з неї. І, більше того, ресурси, що вважалися доступними, можуть несподівано пропасти (плотер може зламатися). Таким чином, на практиці лише небагато систем, якщо такі взагалі є, використовують алгоритм банкіра для ухилення від взаємних блокувань.

Алгоритм банкіра представляє для нас інтерес тому, що він дає можливість розподіляти ресурси так, щоб обходити тупикові ситуації. Проте у цього алгоритму є **ряд серйозних недоліків**, із-за яких розробник системи може виявитися змушеним вибрати інший підхід до вирішення проблеми тупиків.

1. Алгоритм банкіра виходить з фіксованої кількості розподілюваних ресурсів. Оскільки пристрої, що представляють ресурси, частенько вимагають технічного обслуговування або із-за виникнення несправностей, або з метою профілактики, ми не можемо вважати, що кількість ресурсів завжди залишається фіксованою.

2. Алгоритм вимагає, щоб число працюючих користувачів (процесів) залишалося постійним. Подібна вимога також є нереалістичною. У сучасних мультипрограмних системах кількість працюючих користувачів увесь час міняється.

Наприклад, велика система з розподілом часу цілком може обслуговувати 100 або більше користувачів одночасно. Проте поточне число обслуговуваних користувачів безперервно міняється, можливо, дуже часто, кожні декілька секунд.

3. Алгоритм вимагає, щоб клієнти (тобто завдання або процеси) гарантовано «платили борги» (повертали виділені їм ресурси) впродовж деякого кінцевого часу. І знову-таки для реальних систем потрібно набагато конкретніші гарантії.

4. Алгоритм вимагає, щоб користувачі заздалегідь вказували свої максимальні потреби в ресурсах. У міру того як розподіл ресурсів стає усе більш динамічним, все важче оцінювати максимальні потреби користувача.