9.2 ВИЯВЛЕННЯ ВЗАЄМНИХ БЛОКУВАНЬ

При використанні технології виявлення і відновлення взаємних блокувань система не намагається уникнути взаємних блокувань. Вона дозволяє їм статися, намагається виявити момент їх виникнення, а потім робить деякі дії з відновлення працездатності. У цьому розділі будуть розглянуті деякі способи виявлення взаємних блокувань і деякі методи відновлення працездатності, які можна реалізувати.

9.2.1 Використання одного ресурсу кожного типу

Розпочнемо з найпростішого випадку, коли використовується тільки один ресурс кожного типу. У системи може бути один сканер, один привід компакт- дисків, один плотер і один накопичувач на магнітній стрічці, але тільки по одному екземпляру кожного класу ресурсів. Іншими словами, ми тимчасово виключаємо системи, які, наприклад, мають два принтери. Вони будуть розглянуті пізніше, з використанням іншого методу.

Для такої системи можна побудувати ресурсний граф, показаний на рис. 9.3.

Якщо цей граф містить один і більше циклів, значить ми маємо справу зі взаємним блокуванням. Будь-який процес, що є частиною циклу, буде заблокований. Якщо циклів немає, значить, система не знаходиться в стані взаємного блокування.

Як приклад складнішої системи порівняно з раніше розглянутими, візьмемо систему з сімома процесами від A до G, і шістьма ресурсами від R до W. Кожен ресурс знаходиться в стані поточної зайнятості, і на кожен ресурс в даний момент поступив запит:

1. Процес A утримує R і хоче отримати S.
2. Процес B не утримує ніяких ресурсів, але хоче отримати T.
3. Процес З не утримує ніяких ресурсів, але хоче отримати S.
4. Процес D утримує U і хоче отримати S і T.
5. Процес E утримує T і хоче отримати V.
6. Процес F утримує W і хоче отримати S.
7. Процес G утримує V і хоче отримати U.

Виникає питання: «Чи знаходиться ця система в стані взаємного блокування, і якщо знаходиться, то які процеси залучені в цей стан»?. Щоб відповісти на це

питання, можна побудувати граф ресурсів і процесів, показаний на рис. 9.4. Цей граф містить один цикл, який можна виявити візуально. Цей цикл показаний на рис. 9.4, б. З циклу видно, що процеси D, E і G залучені у взаємному блокуванні. Процеси A, C і F не знаходяться в стані взаємного блокування, оскільки ресурс S може бути виділений будь-якому з них, який потім закінчить свою роботу і поверне ресурс. Два процеси, які залишилися, зможуть узяти його по черзі і також завершити свою роботу.

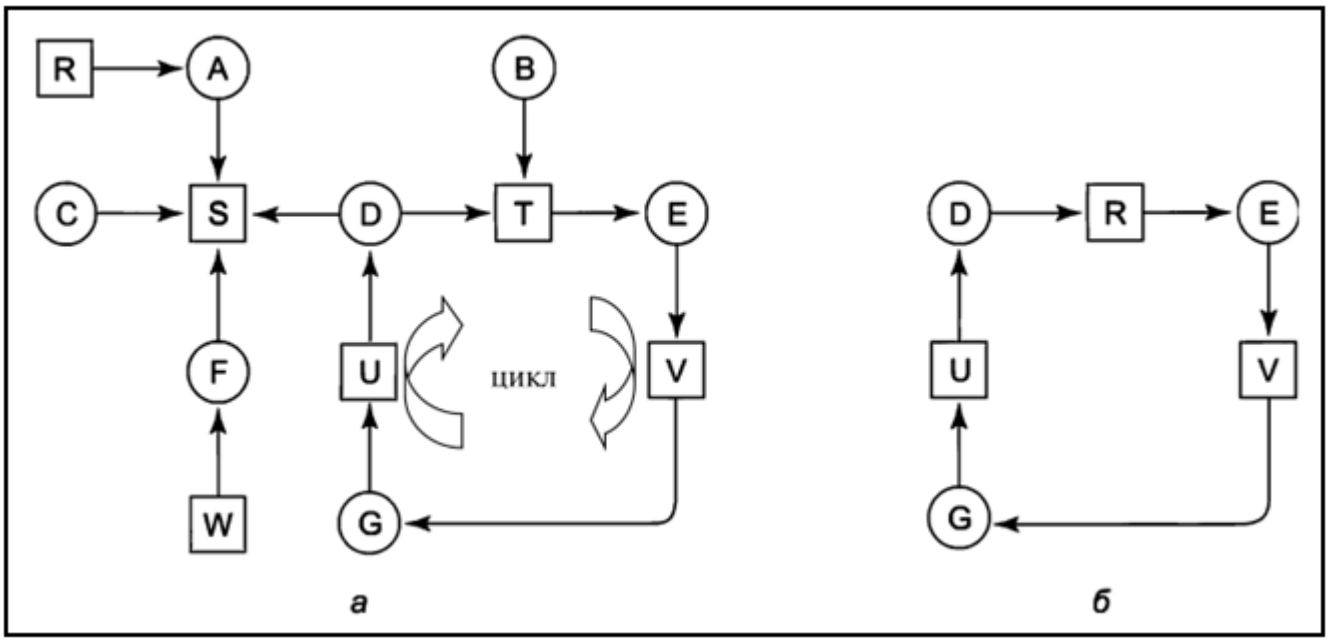

Рисунок 9.4 – Граф ресурсів і процесів (а); витягнутий з нього цикл (б)

Хоча візуально виділити взаємне блокування з простого графа відносно неважко, для використання в справжніх системах потрібний формальний алгоритм виявлення взаємних блокувань. Відомо багато алгоритмів для виявлення циклів у спрямованих графах. Далі буде наведений простий алгоритм, який перевіряє граф і який припиняє свою роботу або при виявленні циклу, або при виявленні відсутності циклів. В алгоритмі використовується одна динамічна структура даних L, що є списком вузлів, а також списком ребер. У процесі роботи алгоритму ребра будуть відмічатися для позначення того, що вони вже були перевірені, щоб запобігти повторним перевіркам.

Дія алгоритму заснована на виконанні таких кроків:

1. Для кожного вузла N, наявного в графі, виконуються наступні п'ять кроків, що використовують вузол N в якості початкового.

2. Ініціалізувався (очищається) список L, а з усіх ребер знімаються позначки.

3. Поточний вузол додається до кінця списку L, і проводиться перевірка, чи не з'явиться цей вузол в списку L двічі. Якщо це станеться, значить, граф містить цикл (відображений в списку L) і алгоритм припиняє роботу.

4. Для заданого вузла визначається, чи немає яких-небудь непомічених ребер, що відходять від нього. Якщо такі ребра є, здійснюється перехід до кроку 5, якщо їх немає, здійснюється перехід до кроку 6.

5. Довільно вибирається і позначається непомічене ребро, що відходить від вузла. Потім по ньому здійснюється перехід до нового поточного вузла, і алгоритм повертається до кроку 3.

6. Якщо цей вузол є первинним вузлом, значить, граф не містить ніяких циклів, і алгоритм завершує свою роботу. Інакше алгоритм зайшов у тупик. Цей вузол видаляється, і алгоритм повертається до попереднього вузла, тобто до того вузла, який був поточним перед тільки що видаленим вузлом, цей вузол робиться поточним і здійснюється перехід до кроку 3.

Цей алгоритм бере по черзі кожен вузол в якості кореневого, в надії, що з цього вийде дерево, і виконує в дереві пошук у глибину. Якщо в процесі обходу алгоритм обходить усі ребра з якого-небудь заданого вузла і повертається до вузла, що вже зустрічався, значить, він знайшов цикл. Якщо він повертається до кореневого вузла і не може йти далі, то підграф, доступний з поточного вузла, не містить циклів. Якщо ця властивість зберігається для всіх вузлів, то це означає, що повний граф не містить циклів, а система не знаходиться в стані взаємного блокування.

Щоб побачити на практиці роботу цього алгоритму, скористаємося графом на рис. 9.4, а. Порядок обробки вузлів довільний, тому досліджуватимемо їх зліва направо і зверху вниз, вибравши при першому запуску алгоритму початковий вузол R, потім послідовно вибираючи вузли A, B, C, S, D, T, E, F і т. д. Якщо ми виявимо цикл, алгоритм припинить свою роботу.

Розпочинаємо з вузла R і ініціалізуємо L як порожній список. Потім додаємо вузол R в список, переходимо до єдино можливого вузла A і додаємо його також до списку L, отримуючи L =[R, A]. З вузла A йдемо до вузла S, отримуючи L =[R, A, S].

Вузол S не має ребер, що відходять від нього, отже, це тупик, який примушує нас повернутися до вузла A. Оскільки у вузла A також немає немаркованих ребер, що відходять від нього, ми повертаємося до вузла R, завершуючи, таким чином, його дослідження.

Тепер перезапускаємо алгоритм, розпочинаючи його роботу з вузла A, заздалегідь повернувши список L в початковий стан. Цей пошук також швидко зупиниться, тому розпочнемо знову з вузла B. З вузла B пройдемо по ребрах, що відходять, до тих пір, поки не дістанемося до вузла D. До цього моменту список матиме такий вигляд: L =[B, T, E, V, G, U, D]. Тепер треба зробити довільний вибір. Якщо вибрати вузол S, ми потрапляємо в тупик і повертаємося до вузла D. Удруге вибираємо вузол T і оновлюємо список L до виду [B, T, E, V, G, U, D, T], де виявляємо цикл і зупиняємо роботу алгоритму.

Цей алгоритм ще далекий від оптимального. Проте наведений приклад доводить саме існування алгоритму для виявлення взаємних блокувань.

9.2.2 Використання декількох ресурсів кожного типу

Коли в системі існує декілька примірників яких-небудь ресурсів, то для виявлення взаємних блокувань потрібний інший підхід. Зараз буде представлений алгоритм (алгоритм Медніка) [29], заснований на використанні матриць і призначений для виявлення взаємних блокувань при роботі n процесів, від P1 до Pn. Нехай m – це число класів ресурсів, E1 – кількість ресурсів класу 1, E2 – кількість ресурсів класу 2, а загалом, Ei – кількість ресурсів класу i (де $1 \leq i \leq m$). E – це вектор існуючих ресурсів. Він передає загальну кількість примірників кожного ресурсу, що є в наявності. Наприклад, якщо клас 1 є накопичувачами на магнітних стрічках, то E1 = 2 означає, що в системі є два такі накопичувачі.

У будь-який момент часу якісь ресурси можуть бути виділені і недоступні. Нехай A буде вектором доступних ресурсів, де Ai дає кількість примірників ресурсу i, доступних на даний момент (тобто не виділених). Якщо обидва накопичувачі на магнітній стрічці вже виділені, A1 дорівнюватиме 0.

Тепер нам потрібні два масиви: C – матриця поточного розподілу і R – матриця запитів, i-й рядок в матриці C говорить про те, скільки примірників кожного класу

ресурсів в даний момент утримує процес Pi. Таким чином, Cij – це кількість примірників ресурсу j, яка утримується процесом i. За аналогією з цим, Rij – це кількість примірників ресурсу j, яку хоче отримати процес Pi. Усі чотири структури даних показані на рис. 9.5.

Для цих чотирьох структур даних зберігається одне важливе співвідношення – кожен ресурс є або виділеним, або доступним. Це спостереження означає, що (i = 1, 2,., n; j = 1,2, ., m):

$$\sum_{i=1}^{n} Cij + Aj = Ej$$

Іншими словами, якщо скласти усі вже виділені екземпляри ресурсу j і до них додати все ще доступні примірники, в результаті вийде кількість існуючих примірників ресурсу цього класу.

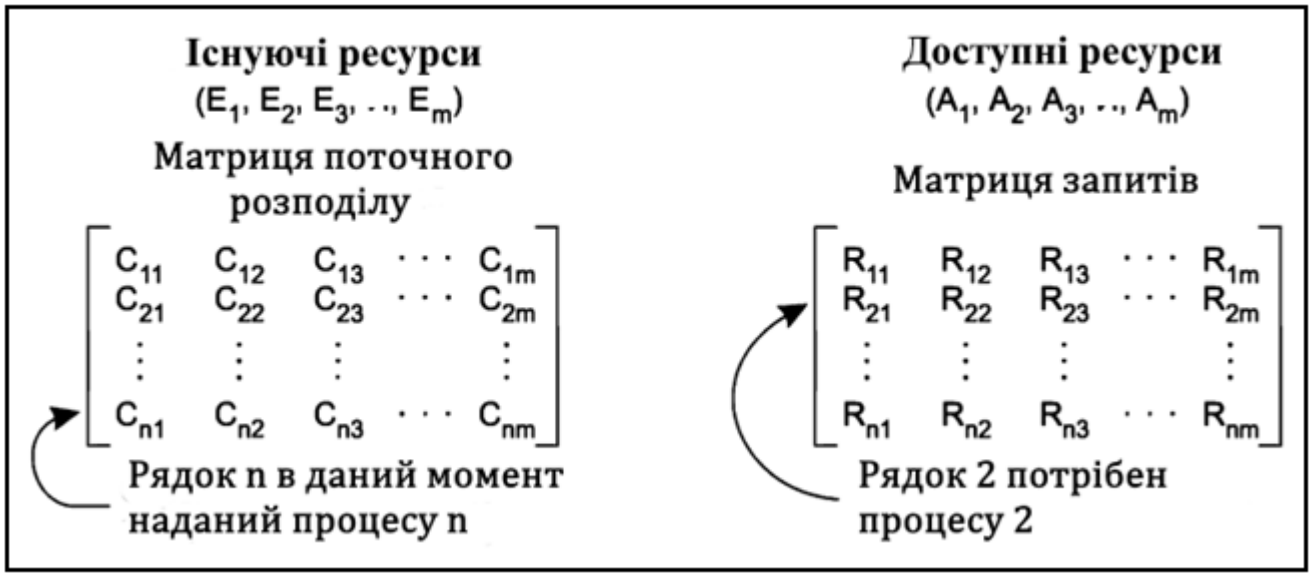

Рисунок 9.5 – Чотири структури даних, необхідні для роботи алгоритму виявлення взаємних блокувань

Алгоритм виявлення взаємних блокувань заснований на порівнянні векторів. Визначимо, що для двох векторів A і B відношення A ≤ B означає, що кожен елемент вектору A менше або дорівнює відповідному елементу вектору B. Математично це можна записати так: A ≤ B тоді і тільки тоді, коли Ai ≤ Bi для 1 ≤ i ≤ m.

Кожен процес спочатку оголошується немаркованим. У міру роботи процеси позначатимуться, показуючи, що вони здатні завершити свою роботу і не беруть участь у взаємних блокуваннях. Коли алгоритм завершує свою роботу, будь-який

немаркований процес вважається таким, що бере участь у взаємному блокуванні. При роботі цього алгоритму передбачається найгірший з можливих сценаріїв розвитку подій: усі процеси утримують усі отримані ресурси до тих пір, поки не закінчать свою роботу.

Тепер алгоритм виявлення взаємного блокування можна викласти в такій послідовності:

1. Пошук немаркованого процесу, Pi, для якого i-й рядок матриці R менше або рівний A.
2. Якщо такий процес знайдений, додаємо до вектору A i-й рядок матриці C (повертаємо зайняті ресурси), встановлюємо мітку на процес і повертаємося до кроку .
3. Якщо такого процесу немає, алгоритм завершує роботу.

Після закінчення роботи алгоритму усі немарковані процеси, якщо такі є, вважаються такими, що беруть участь у взаємних блокуваннях.

На першому кроці алгоритм шукає процес, який може допрацювати до кінця. Такий процес характеризується тим, що всі його запити на ресурси можуть бути задоволені за рахунок поточних доступних ресурсів. Тоді вибраний процес допрацює до кінця, після чого поверне всі утримувані ним ресурси до фонду доступних ресурсів. Потім цей процес позначається завершеним. Якщо в результаті виявиться, що всі процеси можуть допрацювати до кінця, що жоден з них не бере участі у взаємних блокуваннях. Якщо частина процесів ніколи не зможе допрацювати до кінця, значить, вони знаходяться в стані взаємних блокувань. Хоча алгоритм не є детермінованим (оскільки він може запускати процеси в будь-якому можливому порядку), результат завжди буде однаковий.

Розглянемо приклад роботи алгоритму виявлення взаємних блокувань, який показаний на рис. 9.6. Тут зображені три процеси і чотири класи ресурсів, які ми довільно позначили як диски, плотери, сканери і приводи компакт-дисків. Процес 1 утримує один сканер. Процес 2 утримує два диска і один привід компакт-дисків. Процес 3 утримує плотер і два сканери. Кожен процес потребує додаткових ресурсів, що відображено в матриці R.

Під час роботи алгоритму виявлення взаємних блокувань здійснюється пошук процесу, чий запит на ресурс може бути задоволений. Вимоги першого процесу задовольнити неможливо через відсутність доступного приводу компакт-дисків. Запит другого процесу також не можна задовольнити, оскільки немає вільного сканера. Можна задовольнити запит третього процесу, тому третій процес запускається і вивільняє всі ресурси, що утримувалися ним, внаслідок чого виходить, що A=(2 2 2 0).

Рисунок 9.6 – Приклад, що демонструє роботу алгоритму виявлення взаємних блокувань

Тепер може бути запущений процес 2, що вивільняє утримувані ним ресурси, внаслідок чого виходить, що A=(4 2 2 1) і може бути запущений процес, що залишився. При цьому взаємних блокувань у системі не виникає.

Розглянемо незначну зміну ситуації, показаної на рис. 9.6. Припустимо, що процес 2 потребує приводу компакт-дисків, а також двох дисків і плотера. Жоден з цих запитів не може бути задоволений, тому уся система увійде до стану взаємного блокування.

Тепер, коли ми знаємо, як можна виявити взаємне блокування (принаймні, при заздалегідь відомих статичних запитах на виділення ресурсів), виникає питання, коли саме треба приступати до їх пошуку. Тобто, як часто слід виконувати перевірку тупика?

Можна перевірку при видачі кожного запиту на виділення ресурсу. Тим самим буде забезпечено їх виявлення на найранішій стадії, але це занадто обтяжливо для центрального процесора.

У деяких реалізаціях ОС застосовує ще «ледачішу» політику. Незадоволений запит може призвести до тупику, але може і не призвести. ОС не поспішає виконувати перевірку навіть при надходженні такого запиту, а вичікує деякий час: може бути «все само собою улагодиться» – і в більшості випадків саме так і трапляється. І тільки якщо запит залишається незадоволеним впродовж певного часу, ОС береться за пошук тупику. Розмір часової затримки може визначатися швидкісними характеристиками запрошеного ресурсу.

Є й інші альтернативні стратегії, що передбачають перевірку кожні k хвилин, або тільки в тому випадку, якщо міра завантаженості процесора знижується відносно якогось порогу.

Якщо тупик виявлений, то як його ліквідовувати? На жаль, розв'язка тупика практично завжди пов'язана з втратами. Єдиним реальним способом розв'язки є примусове припинення одного або декількох процесів і звільнення утримуваних ними ресурсів. Як вибрати процес-жертву для припинення його роботи?

По-перше, ОС має бути упевнена в тому, що при припиненні вибраних процесів звільниться об'єм ресурсів, достатній для виходу із тупика. По-друге, оцінюється об'єм втрат, пов'язаних з припиненням того або іншого процесу.

Припинений процес, швидше за все, буде запущений повторно. Таким чином, ресурси, використані ним при його незавершеному виконанні, складають прямі втрати. Тому природним рішенням видається припинення того процесу, який до цього моменту використав менше всього ресурсів (не лише монопольних, а будь-яких ресурсів взагалі). Оскільки найдорожчим ресурсом є процесорний час, то вибір жертви за критерієм мінімального використаного часу робиться найчастіше.

Бажано, щоб «постраждалий» процес був знову запущений, причому, можливо навіть, з підвищеним пріоритетом. Але чи всякий процес можна перервати на середині, а потім запустити спочатку?

Наприклад, нехай є процес-програма, яка повинна нарахувати внески на 10 банківських рахунків. Ця програма примусово завершується в той момент, коли вона

обробила тільки 5 рахунків. Якщо при перезапуску програма почне виконуватися з початку, вона повторно нарахує внески на перші 5 рахунків. ОС не може знати, чи призведе перезапуск процесу до небажаних наслідків, тому рішення про повторний запуск перекладається на користувача.

9.2.3 Безпечний і небезпечний стан

При подальшому розгляді алгоритмів ухилення від взаємних блокувань використовується інформація, представлена на рис. 9.5. У будь-який заданий момент часу існує поточний стан структур E, A, C і R. Стан вважається безпечним, якщо існує якийсь порядок планування, при якому кожен процес може допрацювати до кінця, навіть якщо усі процеси несподівано і терміново запросятьмаксимальну

кількість ресурсів. Це положення найпростіше проілюструвати за допомогою прикладу, в якому використовується один ресурс. На рис. 9.7, а показаний стан, в якому процес А утримує 3 примірники ресурсу, але кінець кінцем може зажадати усі 9 примірників. Процес В у цей момент утримує 2 примірника, але пізніше може зажадати в цілому ще 4. Процес С також утримує 2 примірника, але може зажадати ще 5. У системі є всього 10 примірників цього ресурсу, 7 з яких вже розподілені, а 3 поки що вільні.

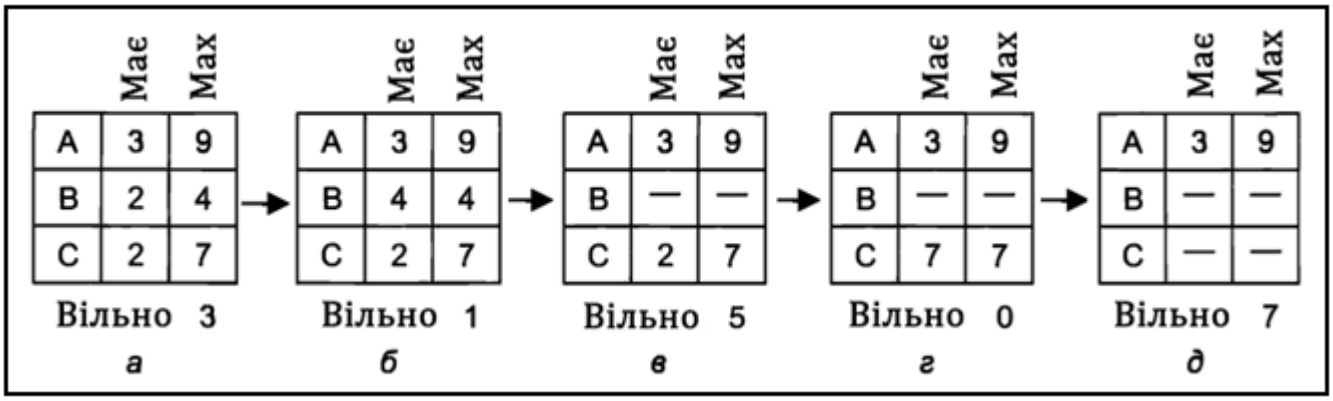

Рисунок 9.7 – Демонстрація того, що стан (а) є безпечним

Стан на рис. 9.7, а є безпечним, тому що існує така послідовність надання ресурсів, яка дозволяє завершитися усім процесам – планувальник може просто запустити в роботу тільки процес на той час, який він запросить і отримає два додаткові примірники ресурсу, що призведе до стану, зображеного на рис. 9.7, б. Коли процес В завершить свою роботу, ми отримаємо стан, показаний на рис. 9.7, в. Потім

планувальник може запустити процес С, що з часом призведе нас до ситуації, показаної на рис. 9.7, г. Після закінчення роботи процесу С отримаємо ситуацію, показану на рис. 9.7, д.. Тепер процес А може отримати необхідні йому шість примірників ресурсу і завершити свою роботу. Таким чином, стан, показаний на рис. 9.7, а є безпечним, оскільки система може уникнути взаємних блокувань за допомогою ретельного планування процесів.

Тепер припустимо, що початковий стан системи показаний на рис. 9.8, а [25]. Але в даний момент процес А просить і отримує ще один ресурс, і система переходить в стан, показаний на рис. 9.8, б. Чи зможемо ми знайти послідовність, яка гарантує безпечну роботу системи?

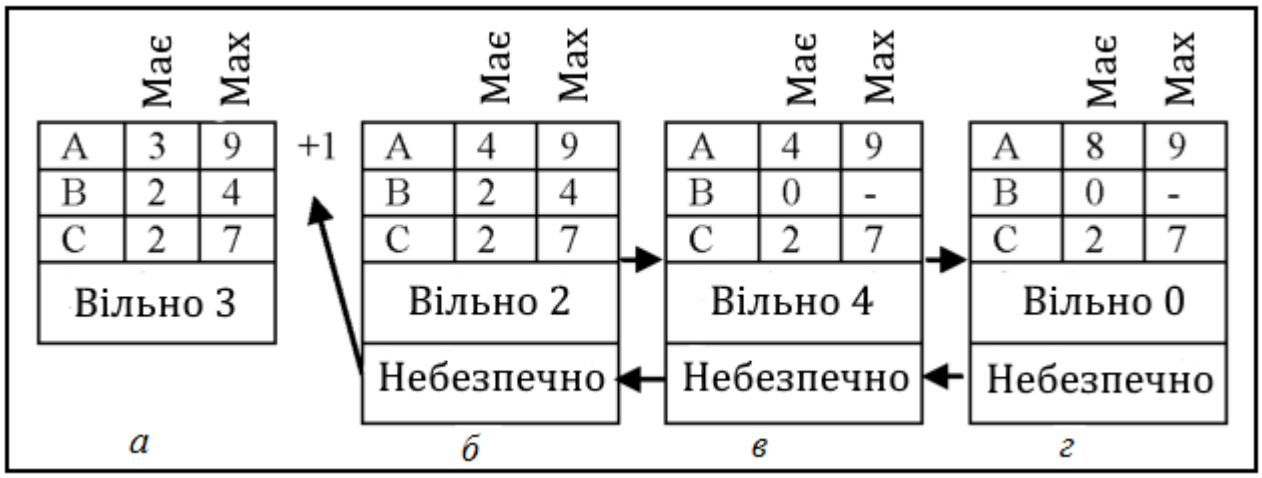

Рисунок 9.8 – Демонстрація того, що стан (б) небезпечний

Планувальник може дати попрацювати процесу В до того моменту, поки він не запросить усі свої ресурси, як показано на рис. 9.8, с. Зрештою процес В успішно завершується, і ми отримуємо ситуацію, показану на рис. 9.8, г. У результаті в системі залишилися тільки чотири вільні примірники ресурсу, а кожному з активних процесів необхідно по п'ять примірників.

Отже, рішення про надання ресурсу, яке перевело систему із стану, показаного на рис. 9.8, а, до стану, показаного на рис. 9.8, б, перевело її з безпечного в небезпечний стан. Якщо із стану, показаного на рис. 9.8, б, запустити процес А або процес С, то жоден з них не запрацює. Повертаючись назад, треба сказати, що запит процесу А не повинен був задовольнятися.

Слід зазначити, що небезпечний стан сам по собі не є станом взаємного блокування. Починаючи із стану, показаного на рис. 9.8, б, система може попрацювати деякий час. Фактично може навіть успішно завершитися робота одного з процесів. Таким чином, різниця між безпечним і небезпечним станами полягає в тому, що в безпечному стані система може гарантувати, що всі процеси закінчать свою роботу, а в небезпечному стані такої гарантії дати не можна.

9.2.4 Реалізація алгоритму виявлення тупика

У загальному випадку для побудови алгоритму виявлення тупика (алгоритм Медніка) використовуватимемо ті ж структури, що визначені вище [29]. 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 *Allocation*(n*m) відображає фактичне виділення системою ресурсів процесам. Якщо *Allocation* [i, j] = k, то процесу i в даний момент виділено системою k одиниць ресурсу j.

Вектор *Requesti* (довжини m) – вектор запитів для процесу Pi. Якщо *Requesti* [j] = k, то процес Pi просить k екземплярів ресурсу Rj.

Алгоритм виявлення тупиків.

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 і 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 і від 1 до n, то система в стані тупика.

Більше того, якщо *Finish[i] = false,* то процес *Pi* – в стані тупика.

Обгрунтування і доказ коректності алгоритму надаємо читачеві.

Приклад використання алгоритму виявлення тупиків.

Нехай є 5 процесів – P0, …, P4, і 3 типи ресурсів – ресурс A (7 примірників), ресурс B (2 примірники) і ресурс C (6 примірників). Нехай стан системи в момент T0 такий:

	Allocation			Request		
	A	**B**	**C**	**A**	**B**	**C**
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	0
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2

У цьому стані системи послідовність процесів <P0, P2, P3, P1, P4> безпечна (перевірте це!).

У продовження прикладу, нехай процес P2 просить додатковий ресурс типу C:

	Request		
	A	**B**	**C**
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

У даному випадку має місце тупик, в якому знаходяться процеси P1, P2, P3, P4. Перевірте це.