## 9.5 ЗАПОБІГАННЯ ВЗАЄМНИХ БЛОКУВАНЬ. ЗАВИСАННЯ

Як все-таки можна уникнути взаємних блокувань в реальних системах? Відхилитися від них по суті неможливо, оскільки для цього потрібна інформація про майбутні запити, про які нічого не відомо. Щоб відповісти на це питання, повернемося назад до чотирьох умов, сформульованих Коффманом та іншими, і подивимося, чи зможуть вони дати нам ключ до вирішення проблеми. Якщо ми зможемо гарантувати, що хоч би одну з цих умов ніколи не буде виконано, то взаємні блокування стануть структурно неможливими.

### 9.5.1 Порушення умови взаємних блокувань

Доступ до деяких ресурсів має бути винятковим. Проте, деякі пристрої вдається усуспільнити. Як приклад розглянемо принтер. Відомо, що намагатися здійснювати виведення на принтер можуть декілька процесів. Щоб уникнути хаосу організовують проміжне формування усіх вихідних даних процесу на диску, тобто пристрої, що розділяється. Лише один системний процес, що називається сервісом або демоном принтера, який відповідає за виведення документів на друк у міру звільнення принтера, реально з ним взаємодіє. Ця схема називається спулінгом (spooling). Таким чином, принтер стає пристроєм, що розділяється, і тупик для нього усунений.

Що вийде, якщо кожен з двох процесів заповнить по половині доступного простору, виділеного на диску під черги на друк своїми вихідними даними, і жоден з них не сформує свої повні вихідні дані? У такому разі ми отримаємо два процеси, що завершили формування тільки частини, але не усього об'єму своїх вихідних даних, і не мають можливості продовжити свою роботу. Жоден з процесів не зможе коли-небудь завершитися, і ми отримаємо взаємне блокування, пов'язане з виведенням даних на диск. Але за наявності дисків великої місткості вичерпання дискового простору стає малоймовірним.

### 9.5.2 Порушення умови утримання і очікування ресурсів

Умови очікування ресурсів можна уникнути, зажадавши виконання стратегії двофазного захоплення. У першій фазі процес повинен просити усі необхідні йому

ресурси відразу. До тих пір, поки вони не надані, процес не може продовжувати виконання.

Якщо в першій фазі деякі ресурси, які були потрібні цьому процесу, вже зайняті іншими процесами, то цей процес повинен звільнити усі ресурси, які були йому виділені, і намагатися повторити першу фазу.

Цей підхід нагадує вимогу захоплення усіх ресурсів заздалегідь. Природно, що тільки спеціально організовані програми можуть бути призупинені впродовж першої фази і рестартовани згодом.

Таким чином, один із способів – змусити усі процеси зажадати потрібні їм ресурси перед виконанням («все або нічого»). Якщо система в змозі виділити процесу все необхідне, він може працювати до завершення. Якщо хоч би один з ресурсів зайнятий, процес чекатиме.

Це рішення застосовується в пакетних системах, які вимагають від користувачів перелічити всі необхідні його програмі ресурси. Проте в цілому подібний підхід не занадто привабливий і призводить до неефективного використання комп'ютера. Як уже відзначалося, перелік майбутніх запитів до ресурсів нечасто вдається спрогнозувати. Якщо така інформація є, то можна скористатися алгоритмом банкіра. Відмітимо також, що описуваний підхід суперечить парадигмі модульності в програмуванні, оскільки додаток повинен знати про передбачувані запити до ресурсів у всіх модулях.

### 9.5.3 Порушення принципу відсутності перерозподілу

Якби можна було відбирати ресурси в процесів до завершення їх роботи, то вдалося б добитися невиконання третьої умови виникнення взаємних блокувань. Перелічимо мінуси цього підходу.

По-перше, відбирати в процесів можна тільки ті ресурси, стан яких легко зберегти, а пізніше відновити, наприклад стан процесора. По-друге, якщо процес впродовж деякого часу використовує певні ресурси, а потім звільняє ці ресурси, то він може втратити результати роботи, виконаної за цей час. Нарешті, наслідком цієї схеми може бути дискримінація окремих процесів, в яких постійно відбирають ресурси.

Питання в ціні подібного рішення, яка може бути занадто високою, якщо необхідність відбирати ресурси виникає часто.

### 9.5.4 Порушення умови кругового очікування

Важко запропонувати розумну стратегію, щоб уникнути останньої умови з розділу «Умови виникнення тупиків» – циклічного очікування.

Один із способів уникнути умови кругового очікування – діяти відповідно до правила, згідно з яким кожен процес може мати тільки один ресурс в кожен момент часу. Якщо потрібний другий ресурс – звільни перший. Очевидно, що для багатьох процесів це неприйнятно.

Інший спосіб – упорядкувати ресурси. Наприклад, можна присвоїти усім ресурсам унікальні номери і зажадати, щоб процеси просили ресурси в порядку їх зростання. Тоді кругове очікування виникнути не може. Наприклад, якщо процес запросив ресурс R1, то далі він може запросити тільки такі ресурси, які упорядкувани за значенням R1. Якби в прикладі, наведеному на рис. 9.1, потоки A і B замовляли ресурси в однаковому порядку (порт, диск), то взаємного блокування можна було б уникнути.

Нехай, наприклад, всі ресурси повністю впорядковані від 1 до r. Ми можемо накласти наступне обмеження: процес не може запитувати ресурс Rk, якщо він утримує ресурс Rh і при цьому $k < h$.

Видно, що, використовуючи це правило, ми ніколи не будемо входити в тупики. Наведемо приклад того, як застосовується це правило.

Нехай є процес, який використовує ресурси, впорядковані як A, B, C, D, E, в такий спосіб (рис. 9.11):

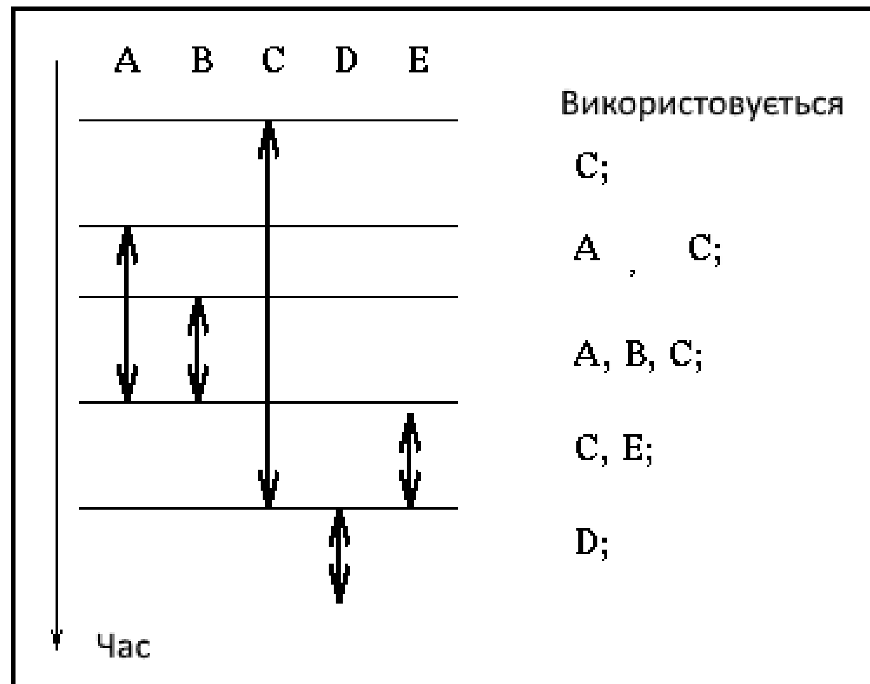


Рисунок 9.11 – Приклад впорядкування ресурсів

Тоді процес може робити наступне:

- захопити (A); захопити (B); захопити (C);
- використати C;
- використати A, C;
- використати A, B, C;
- звільнити (A), звільнити (B), захопити (E);
- використати C і E;
- звільнити (C), звільнити (E), захопити (D);
- використати D;
- звільнити (D).

Стратегія цього типу може використовуватися, коли ми маємо декілька ресурсів.

## ЗАВИСАННЯ

Проблемою, тісно пов'язаною із взаємним блокуванням, є зависання. У динамічній системі запит ресурсів відбувається постійно. Для того щоб прийняти рішення, хто, коли і який ресурс отримає, потрібна певна політика. Ця політика, навіть і розумна, може призвести до того, що деякі процеси ніколи не будуть обслужені, навіть якщо вони не знаходяться в стані взаємного блокування.

Як приклад розглянемо розподіл принтера. Уявимо собі, що система використовує деякий алгоритм, який гарантує, що розподіл принтера не призводить до взаємоблокування. Тепер припустимо, що декілька процесів разом захотіли отримати принтер у своє розпорядження. І хто його отримає?

Один з можливих алгоритмів передбачає передачу принтера тому процесу, в якого, наприклад, найменший файл для виведення на друк. Такий підхід до максимуму збільшує число щасливих клієнтів і представляється цілком справедливим. А тепер подивимося, що вийде на працюючій системі, де в одного процесу є для виведення на друк величезний файл. Як тільки принтер звільниться в черговий раз, система може вибрати знову процес з найкоротшим файлом. Якщо потік процесів з короткими файлами не вичерпується, процес з величезним файлом не отримає принтер ніколи. Він просто намертво зависне (буде відкладений назавжди, навіть якщо і не буде заблокований).

Зависання можна уникнути за рахунок використання політики розподілу ресурсів «першим прийшов – першим і обслужений». При такому підході процес, який очікує довше всіх, обслуговується наступним. Зрештою, будь-який заданий процес з часом стане найстаршим у черзі і отримає необхідний йому ресурс.

## ТУПИКИ В СИСТЕМАХ СПУЛІНГУ

Системи спулінга часто виявляються схильні до тупиків. Режим спулінга (введення-виведення з буферизацією) застосовується для підвищення продуктивності системи шляхом ізолювання програми від такого низькошвидкісного периферійного устаткування, як пристрій виведення даних на принтер. Якщо, наприклад, програмі, що видає рядки даних на принтер, доводиться чекати роздруку кожного рядка перед передачею наступного рядка, то така програма виконуватиметься повільно. Щоб підвищити швидкість виконання програми, рядки даних, призначені для друку, спочатку записуються на більше високошвидкісний пристрій, наприклад дисковий накопичувач, де вони тимчасово зберігаються до моменту роздруку.

У деяких системах спулінга програма повинна сформувати усі вихідні дані – тільки після цього починається реальний роздрук. У зв'язку з цим декілька незавершених завдань, що формують рядки даних і записують їх у файл спулінга для друку, можуть опинитися в тупиковій ситуації, якщо передбачена місткість буфера буде заповнена до того, як завершиться виконання якого-небудь завдання. Для відновлення, або виходу з подібної тупикової ситуації, міг би знадобитися перезапуск, рестарт системи з втратою всієї роботи, виконаної до цього моменту.

Якщо система потрапляє в тупикову ситуацію таким чином, що в оператора ЕОМ залишаються можливості управління, то в якості менш радикального способу відновлення працездатності можна запропонувати знищення одного або декількох завдань, поки у завдань, що залишаються, не виявиться достатньо вільного місця в буфері, щоб вони могли завершитися.

Коли системний програміст робить генерацію (чи налаштування) операційної системи, він задає розмір буферних файлів для спулінга. Один із способів зменшити ймовірність тупика при спулінгу полягає в тому, щоб передбачити значно більше місця для файлів спулінга, чим знадобиться згідно з попередньою оцінкою. Подібне рішення не завжди здійсненно, якщо пам'ять дефіцитна.

Поширеніше рішення полягає в тому, що для процесів вхідного спулінга встановлюються обмеження, з тим щоб вони не могли приймати додаткові завдання, коли файли спулінга починають наближатися до деякого порогу насичення, наприклад, виявляються заповненими на 75 відсотків. Такий підхід може призвести

до деякого зниження продуктивності системи, проте це – та ціна, яку доводиться платити за зменшення ймовірності тупика.

Сучасні системи є в цьому сенсі набагато досконалішими. Вони можуть дозволяти починати роздрук до того, як завершиться чергове завдання, з тим щоб повний або майже повний файл спулінга спустошився повністю або частково вже в процесі виконання завдання. У багатьох системах передбачається динамічний розподіл буферної пам'яті, так що, якщо відведеного місця в пам'яті виявляється мало, для файлів спулінга виділяється додаткова пам'ять.