**1. Які основні примітиви синхронізації ви можете перелічити? Які з них**

**наявні в обраній вами мові програмування?**

**1. М'ютекси (Mutexes)**: Це примітиви синхронізації, які дозволяють тільки одному потоку використовувати ресурс (наприклад, змінну або область пам'яті) в певний момент часу. В С++, м'ютекси надаються стандартною бібліотекою `std::mutex`.

**2. Семафори (Semaphores):** Це засоби синхронізації, які дозволяють обмежити кількість потоків, які можуть одночасно використовувати деякий ресурс. Семафори можна використовувати для управління доступом до обмежених ресурсів. У С++ семафори можна реалізувати за допомогою `std::semaphore` або за допомогою механізмів, наданих платформою, таких як POSIX-семафори.

**3. Умовні змінні** (Condition Variables): Це примітиви синхронізації, які дозволяють потокам чекати на певну умову, перш ніж продовжити виконання. Умовні змінні дозволяють ефективно просипати потоки, які чекають на певну умову. В С++, умовні змінні надаються за допомогою `std::condition\_variable`.

**4. Атомарні операції** (Atomic Operations): Це операції, які гарантують атомарність виконання безпеки відносно конкурентних потоків. У С++, атомарні операції надаються за допомогою класу `std::atomic`.

**2. Які додаткові примітиви синхронізації ви можете перелічити? Чи знаєте щось про згадані в теоретичному матеріалі до роботи додаткові примітиви?**

спін-лок (spin lock), рекурсивний лок (recursive lock), рід-райт

лок (read-write lock, в джерелах зустрічається також як shared-unique).

**3. Що таке м’ютекс? Чому використання (взяття, блокування) м’ютексувважається важкою операцією? Специфічні для конкретних ОС (Windows,Unix) елементи та особливості синхронізації.**

М’ютекс (Mutex) – примітив синхронізації, головною задачею якого є

обмеження одночасної кількості потоків виконання всередині процесу,

що мають доступ до певної ділянки коду. М’ютекс обмежує доступ до

ділянки коду лише для одного потоку виконання. Класичний інтерфейс

м’ютекса складається з двох методів (або функцій, в залежності від

мови): lock та unlock. Потік виконання, що викликає метод lock, стає

власником м’ютекса, і допоки власник не викликає метод unlock

(звільнить м’ютекс), виклики до методу lock з інших потоків змусять їх

чекати виклику unlock від потоку власника. Виклик методу lock два

рази підряд з одного потоку виконання в більшості випадків приводить

до помилки часу виконання.

1. Блокування та Взяття: Взяття м'ютексу (lock) означає, що потік намагається отримати доступ до захищеного м'ютексом ресурсу, а якщо м'ютекс вже використовується іншим потоком, цей потік буде заблокований до того часу, поки м'ютекс не буде звільнений. Ця операція може бути важкою через можливість виникнення ситуації "взаємного блокування" (deadlock), коли два або більше потоки чекають один на одного.

2. Управління Ресурсами: Використання м'ютексу потребує управління ресурсами, що може бути складним завданням, особливо при великій кількості потоків та ресурсів.

3. Потенційна Низька Продуктивність: Багатократне взяття та звільнення м'ютексу може призвести до низької продуктивності, оскільки це операції, які потребують часу.

**4. Що таке взаємне блокування (deadlock)? Як воно виникає? З якими**

**примітивами синхронізації? Найпростіший спосіб (головне правило)**

**уникнення взаємного блокування?**

Взаємне блокування (deadlock) — це ситуація, коли два або більше потоків чекають на ресурс, який утримує інший потік, в той час як вони утримують ресурс, який потрібен для продовження роботи потоку, що утримує перший ресурс. У результаті жоден з потоків не може продовжити свою роботу, тому вони залишаються заблокованими назавжди.

Взаємне блокування може виникати внаслідок неправильного використання примітивів синхронізації, таких як м'ютекси, семафори, умовні змінні тощо. Наприклад, це може статися, коли потік А утримує м'ютекс А і намагається отримати м'ютекс В, тоді як потік В утримує м'ютекс В і намагається отримати м'ютекс А. Жоден з потоків не відпустить свій м'ютекс, тому обидва потоки залишаться заблокованими.

Головне правило для уникнення взаємного блокування - це мати строго визначену послідовність отримання блокування на ресурси та завжди утримувати їх в одній і тій же послідовності. Це відомо як правило взаємного виключення. Наприклад, якщо потік А потребує блокування ресурсу X, а потік Б потребує блокування ресурсу Y, то потік А завжди повинен спочатку захопити ресурс X, а потім захопити ресурс Y, і навпаки для потоку Б. Це правило виключення гарантує, що немає ситуацій, коли один потік утримує ресурс, який потребує інший потік, щоб вивільнити.

**5. Що таке умовна змінна? Чи присутній даний примітив в обраній вами**

**мові програмування? Інтерфейс примітиву, методи wait та notify.**

Умовна змінна (Conditional Variable) – примітив синхронізації,

котрий блокує виконання потоку (або декількох потоків), допоки іншій

потік не виконає певну умову (condition) і не оголосить про це

(повідомить про це потоки виконання, що чекають).

У мові програмування C++ умовні змінні доступні через стандартну бібліотеку `std::condition\_variable`. Вони використовуються разом з м'ютексами для створення потокобезпечних програм.

1. `wait(lock)`: Цей метод блокує потік і чекає на сповіщення (сигнал) від іншого потоку або втрату заблокованого стану. Він приймає об'єкт м'ютекса як параметр, який блокується перед входом в метод `wait`. Після виклику цього методу м'ютекс розблокується і потік засинає, чекаючи на сповіщення або втрату заблокованого стану.

2. `notify\_one()`: Цей метод відправляє сигнал одному з потоків, які чекають на умовну змінну. Який саме потік буде розблокований, невідомо, і це залежить від планувальника потоків.

3. `notify\_all()`: Цей метод відправляє сигнал усім потокам, які чекають на умовну змінну. Всі потоки будуть розблоковані і знову перевірять умову.

**6. Способи зупинки та продовження роботи потоку в обраній мові**

**програмування? Способи реалізації в мовах, які їх не підтримують.**

1. Сплячий режим (Sleeping): Функція `std::this\_thread::sleep\_for` або `std::this\_thread::sleep\_until` дозволяє потоку заснути на певний період часу або до певного моменту часу, відповідно. Це може бути корисно, коли потрібно затримати виконання потоку на певний час.

2. Чекання на завершення іншого потоку (Joining): Метод `join` потоку чекає, доки інший потік не завершить своє виконання.

3. Передача сигналу для завершення (Signaling for Termination): Можна використовувати прапорці або атомарні змінні для індикатора завершення роботи потоку.

**7. Які основні технічні та архітектурні проблеми вирішує пул задач? В**

**варіантах задач використовується метрика у вигляді часу виконання**

**задачі, чи може це бути реальною метрикою в ПЗ, який аналог на вашу**

**думку можна використати в реальному ПЗ?**

Пул задач (task pool або thread pool) - це механізм, що вирішує декілька технічних та архітектурних проблем у багатопотокових програмах. Основні проблеми, які вирішує пул задач, включають:

1. Ефективне використання ресурсів: Пул задач дозволяє ефективно використовувати обмежені ресурси системи, такі як центральний процесор (CPU) та кількість доступних потоків. Він управляє розподілом задач між доступними потоками, забезпечуючи максимальне завантаження процесора та мінімальні затримки.

2. Управління навантаженням: Пул задач дозволяє регулювати навантаження на систему, контролюючи кількість активних потоків та обробку задач. Це допомагає уникнути перевантаження системи та забезпечує стабільну та ефективну роботу програми.

3. Уникнення затримок через створення потоків: Створення та видалення потоків може бути витратним процесом. Використання пула задач дозволяє уникнути цих затримок, оскільки потоки створюються заздалегідь та перевикористовуються для обробки різних задач.

4. Керування конкуренцією: Пул задач робить керування конкуренцією більш простим та передбачуваним. Він дозволяє уникнути ситуацій перевантаження потоків, затримок через блокування та взаємного блокування (deadlock), що часто виникають у складних багатопотокових програмах.

5. Підтримка складних сценаріїв виконання: Пул задач дозволяє легко реалізувати складні сценарії виконання, такі як паралельна обробка декількох завдань, планування пріоритетів, обробка та відновлення помилок та інші.

У цілому, пул задач допомагає оптимізувати роботу багатопотокових програм, забезпечуючи ефективне використання ресурсів, керування навантаженням та забезпечення надійності та стабільності роботи програми.

**8. Яка на вашу думку кількість потоків може вважатися оптимальною для пулу потоків і за яких умов?**

Оптимальна кількість потоків для пулу потоків залежить від конкретної задачі, характеристик обладнання, доступних ресурсів та вимог до продуктивності програми. Тут декілька факторів, які варто враховувати:

1. Кількість процесорних ядер: Оптимальна кількість потоків може збігатися з кількістю доступних процесорних ядер. Якщо кожен потік має власне процесорне ядро для виконання, то використання більшої кількості потоків може покращити продуктивність.

2. Характеристики задач: Якщо задачі, які будуть виконуватися потоками, є важкими та вимагають значних обчислювальних ресурсів, то більша кількість потоків може забезпечити кращу паралельну обробку. Однак для легких задач велика кількість потоків може призвести до зайвого навантаження системи.

3. Підтримка операційної системи: Деякі операційні системи мають обмеження на кількість одночасно активних потоків. Врахуйте ці обмеження при визначенні оптимальної кількості потоків.

4. Інтенсивність вводу/виводу: Якщо програма включає значну кількість операцій вводу/виводу (наприклад, робота з файлами чи мережею), то більша кількість потоків може призвести до зайвого очікування і блокування ресурсів. У таких випадках оптимальна кількість потоків може бути меншою.

5. Моніторинг та оптимізація: Важливо проводити моніторинг роботи програми та оптимізувати кількість потоків на основі даних про використання ресурсів. Регулярне тестування різних конфігурацій допоможе знайти оптимальну кількість потоків для конкретного випадку використання.

Отже, оптимальна кількість потоків для пулу потоків буде залежати від специфіки програми та її вимог до продуктивності.

**9. Як на вашу думку потрібно оброблювати ситуацію, коли задача з пулу**

**потоків, намагається додати іншу задачу в цей же пул потоків?**