Асинхронность и Многопоточность

Лекторы:
Аспирант МФТИ, Шер Артём Владимирович
Аспирант МФТИ, Зингеренко Михаил Владимирович

8 октября 2024

Лямбда функция

Лямбда-функция — это анонимная функция, определяемая прямо в месте использования. Используется для передачи небольших функций как аргументов в другие функции (например, в алгоритмы).

[захват переменных из области видимости] (параметры) спецификаторы $->$ возвращаемое_значение

{

тело_функции;

};

Лямбда код

```
1 auto sum = [](int a, int b) throw() -> int {
2     return a + b;
3 };
4
5 std::cout << sum(3, 5);  // Prints: 8
```

Захват переменных из внешнего контекста:

- $[=]$ — захват по значению.
- $[\&]$ — захват по ссылке.
- $[x, \&y]$ — захват конкретных переменных.

Удобны для коротких операций, таких как сортировка или фильтрация.

В чем разница

- Асинхронность: когда задачи запускаются, выполняются и завершаются в перекрывающиеся временные промежутки.
- Параллелизм: когда две или более задачи выполняются одновременно.

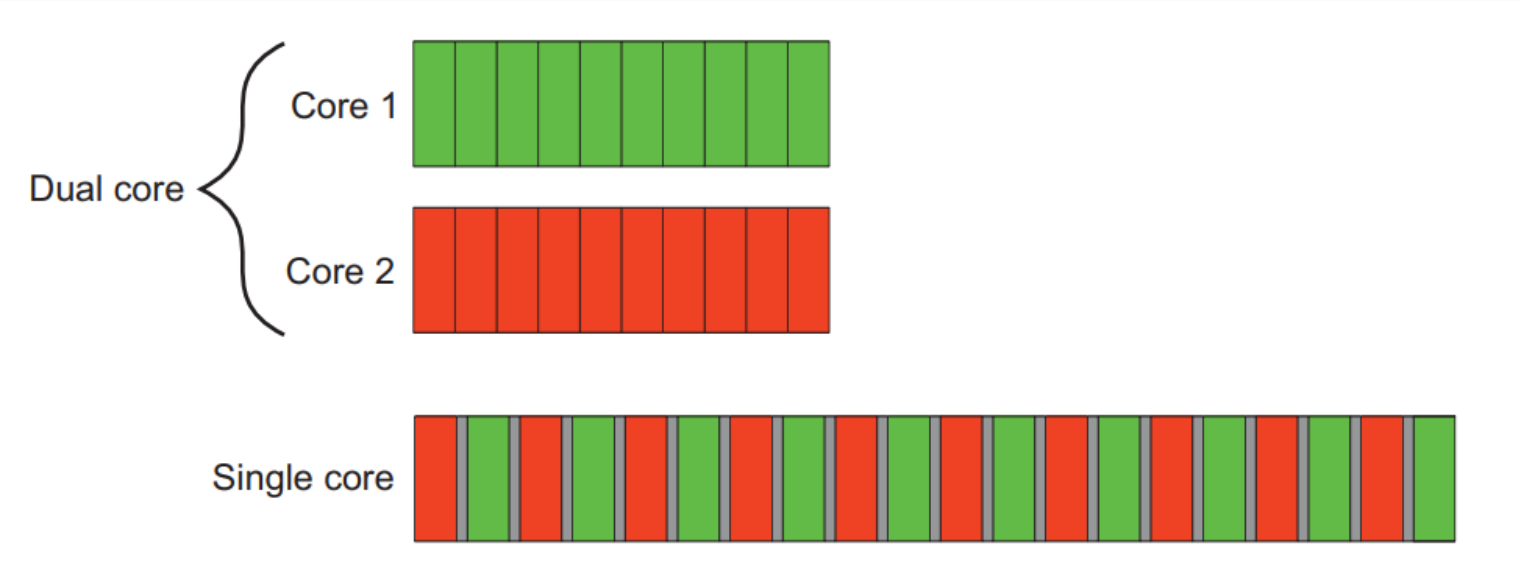
Зачем?

Улучшение производительности.

Снижение времени ожидания пользователя в программах.

Увеличение полезной нагрузки на CPU/GPU

Картиночка



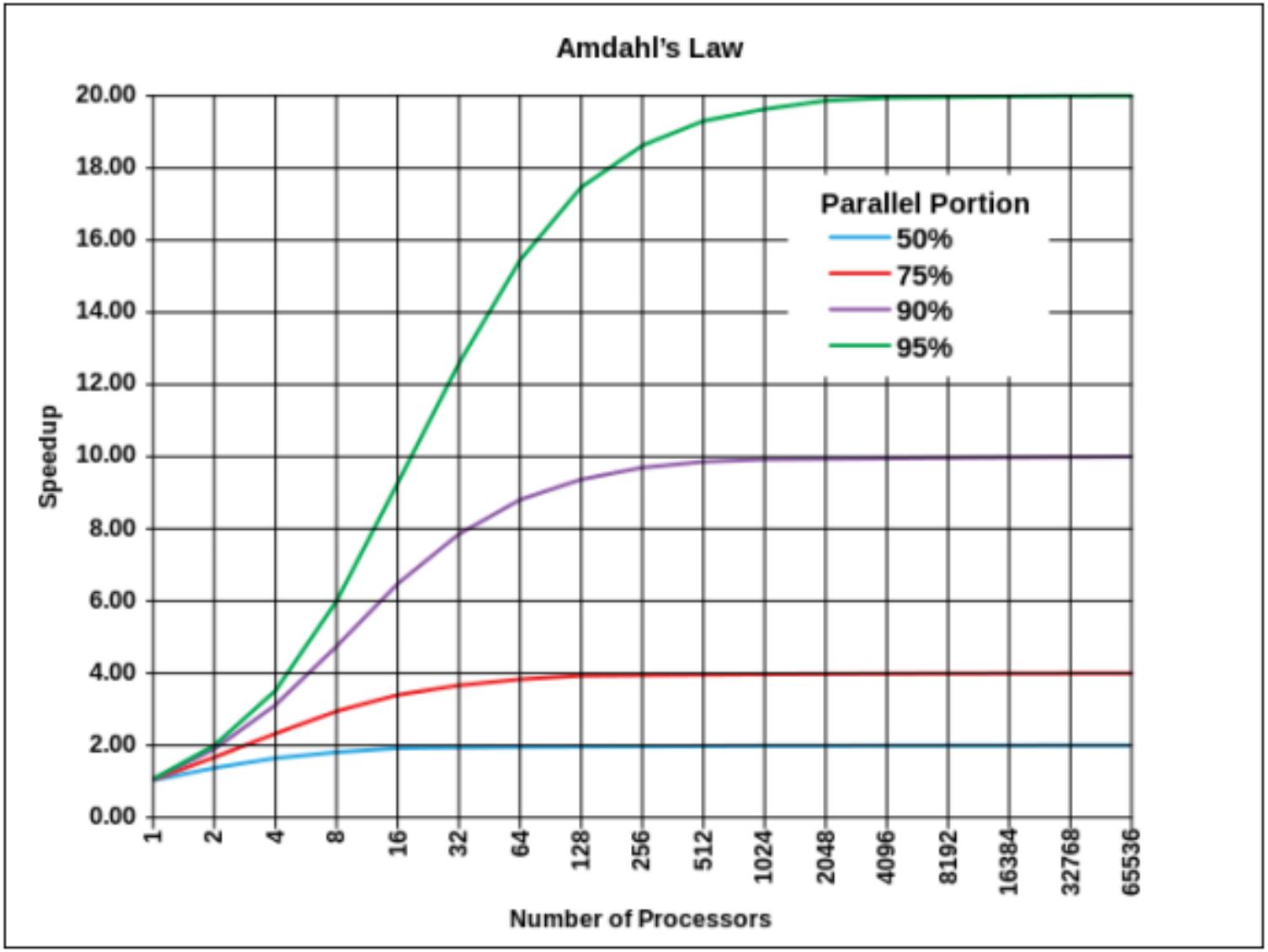
Два подхода к асинхронности: параллельное выполнение на двухъядерном процессоре и переключением задач на одноядерной машине

Закон Амдала

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

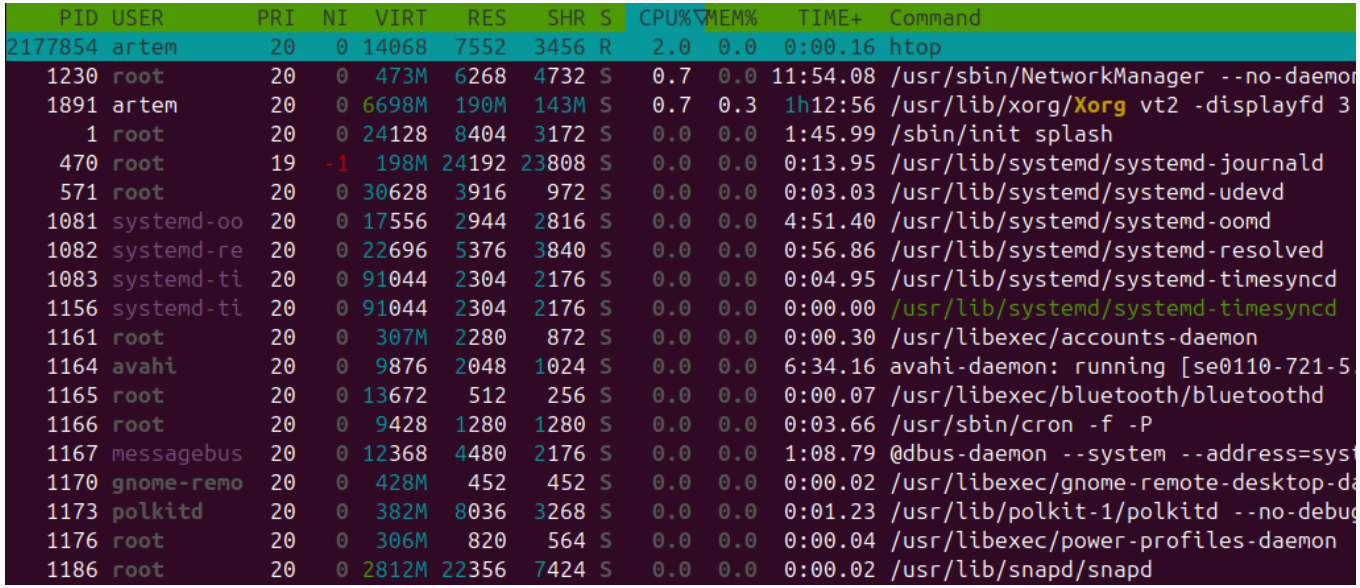
- S – ускорение при использовании N процессоров.
- P – доля программы, которая может быть распараллелена.
- 1 - P – доля программы, которая остаётся последовательной.

Закон Амдала

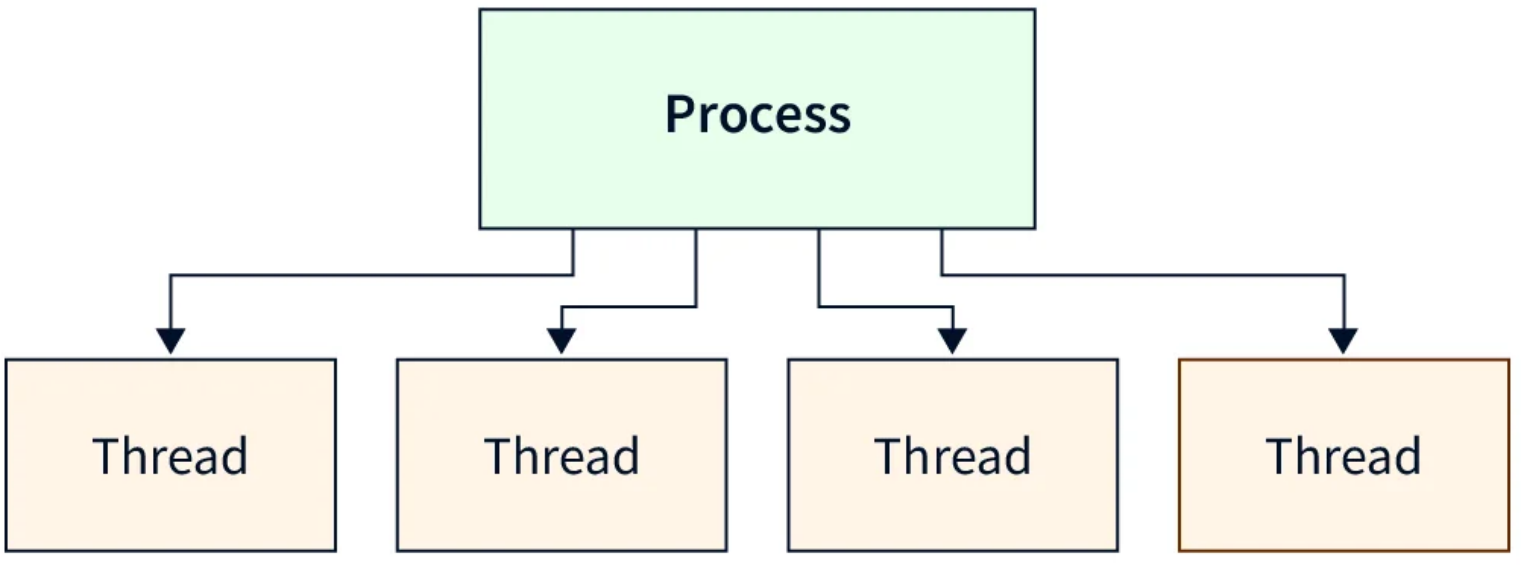


Процессы

Процесс в Linux — это экземпляр запущенной программы в операционной системе Linux. Он представляет собой независимую единицу исполнения, работающую в собственном пространстве памяти и имеющую собственный набор системных ресурсов. Каждому процессу присваивается уникальный идентификатор процесса (PID), который отличает его от других процессов.



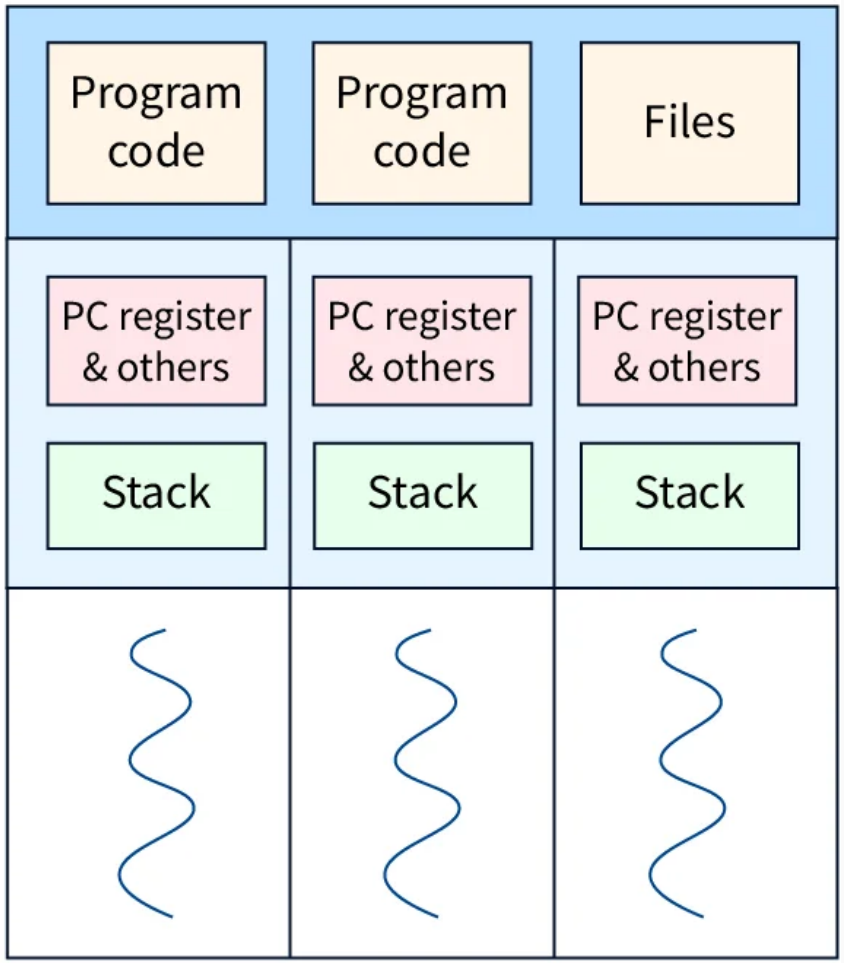
Треды и Процессы



Треды — это легкая единица выполнения внутри процесса, которая может работать независимо. Это позволяет одновременно выполнять задачи, обеспечивая параллельную обработку и эффективное использование ресурсов.

Структура тредов

Внутренняя структура потока состоит из трех основных компонентов: стека, набора регистров и данных, специфичных для потока.



Структура тредов

- Стек — это область памяти, выделенная для выполнения потока. Он содержит локальные переменные потока, вызовы функций и адреса возврата. Каждый поток имеет свой стек, что позволяет выполнять его независимо.
- Набор регистров содержит контекст выполнения потока, включая значения регистров ЦП. В этих регистрах хранится важная информация, такая как счетчики программ, указатели стека и другие регистры, специфичные для процессора.
- Данные, специфичные для потока, позволяют каждому потоку сохранять свое уникальное состояние. Он может включать переменные, специфичные для потока, такие как идентификатор потока или хранилище для конкретного потока.

Создание тредов в C++

```
1  void task1() { /* executing the first task */ }
2  void task2() { /* executing the first task */ }
3
4  int main() {
5      std::thread t1(task1);
6      std::thread t2(task2);
7
8      t1.join();
9      t2.join();
10
11     return 0;
12 }
```

Жизненный цикл треда

- Создание: Поток создается с использованием конструктора std::thread, передавая в него функцию или лямбда-выражение.
- Запуск: Поток начинает выполнять свою задачу сразу после создания.
- Ожидание завершения (join) или отделение (detach): Управление завершением потока.
- Завершение: Поток завершается, когда его функция завершает выполнение.

Асинхронные операции

```cpp
#include <iostream>
#include <future>

int computeSquare(int x) {
    return x * x;
}

int main() {
    std::future<int> result = std::async(std::launch::async, computeSquare, 10);

    std::cout << "Calculating..." << std::endl;
    std::cout << "Result: " << result.get() << std::endl;

    return 0;
}
```

Режимы запуска

- std::launch::async: задача запускается в отдельном потоке.
- std::launch::deferred: задача запускается только при вызове метода get() или wait().

```
1 std::async(std::launch::deferred, computeSquare, 10); // Will be executed when .
      get() is called
2 std::async(std::launch::async, computeSquare, 10);    // Execute immediately in
      a separate thread
```

Packaged task

std::packaged_task – это обёртка для функции или callable-объекта, позволяющая выполнить его асинхронно. Создает объект std::future, который используется для получения результата выполнения задачи. Полезен для работы с задачами в многопоточном окружении.

Зачем использовать std::packaged_task?

- Превращает любую функцию в асинхронную задачу.
- Позволяет передавать задачи в потоки, не теряя возможность управлять результатом.
- Подходит для использования с std::thread или другими пулами потоков.

Packaged task Пример

```cpp
#include <iostream>
#include <future>
#include <thread>

int calculateSquare(int x) {
    return x * x;
}

int main() {
    std::packaged_task<int(int)> task(calculateSquare);
    std::future<int> result = task.get_future();

    std::thread t(std::move(task), 5);
    t.join();

    std::cout << "Square: " << result.get() << std::endl;
    return 0;
}
```

std::packaged_task и std::future работают в связке:

- std::packaged_task отвечает за выполнение задачи.
- std::future позволяет получить результат выполнения.

Можно использовать методы get(), wait(), wait_for() для получения результата.

Асинхронные операции с использованием std::future и std::promise

- std::promise позволяет явно установить результат для std::future
- Используется, когда результат задачи должен быть установлен вручную.

```cpp
#include <iostream>
#include <thread>
#include <future>

void setValue(std::promise<int> p) {
    p.set_value(42);  // Setting a value for future
}

int main() {
    std::promise<int> p;
    std::future<int> f = p.get_future();

    std::thread t(setValue, std::move(p));
    std::cout << "Result from promise: " << f.get() << std::endl;

    t.join();
    return 0;
}
```

Проблемы синхронизации

- Доступ к общим данным: Когда несколько потоков одновременно обращаются к одной и той же переменной или ресурсу без должной синхронизации, это может привести к состояниям гонок (data race).

Примеры

```
1  int counter = 0;
2
3  void increment() {
4      for (int i = 0; i < 1000; ++i) {
5          ++counter;  // Unsynchronized access
6      }
7  }
8
9  int main() {
10     std::thread t1(increment);
11     std::thread t2(increment);
12
13     t1.join();
14     t2.join();
15
16     std::cout << "Counter: " << counter << std::endl;  // Result may be invalid
17     return 0;
18 }
```

Бороться можно с помощью примитивов синхронизации.

Что такое примитивы синхронизации низкого уровня?

Примитивы синхронизации — это базовые конструкции, такие как:

- Мьютекс (Mutex)
- Атомарные операции (Atomic)
- Семафоры (Semaphore)
- Барьеры памяти (Memmory Fence)
- Условные переменные (Condition variable)

Мьютексы

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;
std::mutex mtx;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        ++counter;
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Counter: " << counter << std::endl;
    return 0;
}
```

Мьютексы

В этом примере используется std::lock_guard, который автоматически блокирует мьютекс при создании и разблокирует его при выходе из области видимости. Альтернатива std::unique_lock - более гибкий, позволяет вручную блокировать и разблокировать мьютекс.

```
1 std::unique_lock<std::mutex> lock(mtx);
2 // access to protected data
3 lock.unlock();  // Manual unlock
```

Атомарные операции

std::atomic — это шаблонный класс, предоставляющий атомарные операции с переменными. Гарантирует, что операции над переменной выполняются без гонок данных и не требуют использования мьютексов. Подходит для использования в многопоточных программах, где важно быстро и безопасно изменять значение переменной.

- Атомарные операции:
  - ++var — атомарное увеличение.
  - −−var — атомарное уменьшение.
  - load() и store() — чтение и запись значений.
- Методы fetch:
  - fetch_add(), fetch_sub(), exchange() — более сложные операции.

Пример на атомиках

```cpp
#include <iostream>
#include <atomic>
#include <thread>

std::atomic<int> counter(0);

void increment() {
    for (int i = 0; i < 1000; ++i) {
        ++counter;  // Atomic increment
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);

    t1.join();
    t2.join();

    std::cout << "Counter: " << counter << std::endl;

    counter.fetch_add(10);  // Atomic add 10
    int value = counter.load();  // Atomic read of variable
    return 0;
}
```

Семафоры

Семафор позволяет управлять доступом к ресурсу с определенным количеством доступных слотов.

```cpp
#include <iostream>
#include <thread>
#include <semaphore>

std::counting_semaphore<2> sem(2);  // Two threads can access simultaneously

void criticalSection() {
    sem.acquire();  // Semaphore capture
    std::cout << "Thread in critical section\n";
    std::this_thread::sleep_for(std::chrono::seconds(1));
    sem.release();  // Releasing a semaphore
}

int main() {
    std::thread t1(criticalSection);
    std::thread t2(criticalSection);
    std::thread t3(criticalSection);

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

Conditional variables

std::condition_variable — это механизм синхронизации, который позволяет потокам ожидать наступления определенного события. Используется совместно с мьютексами для организации ожидания и уведомления потоков. Позволяет одному потоку 'усыпить' себя до тех пор, пока другой поток не 'разбудит' его.

- Управляет потоками, ожидающими доступ к ресурсу или наступления условия.
- Позволяет сократить использование активного ожидания (busy-waiting).
- Удобен для реализации паттернов 'производитель-потребитель', задач очередей и синхронизации многопоточных программ.

Conditional variables пример

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
    std::unique_lock<std::mutex> lock(mtx);
    std::cout << "Lock mutex is proceeding...\n";
    cv.wait(lock, [] { return ready; });  // Waiting, until ready does not true
    std::cout << "Worker is proceeding...\n";
}

void setReady() {
    std::lock_guard<std::mutex> lock(mtx);
    std::cout << "Ready is proceeding...\n";
    ready = true;
    cv.notify_one();  // Notification one waiting thread
}

int main() {
    std::thread t1(worker);
    std::this_thread::sleep_for(std::chrono::seconds(1));
    setReady();
    t1.join();
    return 0;
}
```

Conditional variables операции

Основные методы

- wait(): переводит поток в состояние ожидания.
  - wait(lock) — ожидает до вызова notify_one() или notify_all().
  - wait(lock, predicate) — ожидает до тех пор, пока predicate не вернет true.
- notify_one(): пробуждает один из ожидающих потоков.
- notify_all(): пробуждает все ожидающие потоки.

Дэдлок

Дедлок возникает, когда два или более потока навсегда блокируются, ожидая освобождения ресурсов, которые захвачены друг другом. Условия возникновения дедлока:

- Взаимное исключение: один поток удерживает ресурс и не отпускает его.
- Удержание и ожидание: поток удерживает один ресурс и ожидает освобождения другого.
- Отсутствие принудительного освобождения: ресурс не может быть принудительно освобожден.
- Циклическое ожидание: существует цепочка потоков, каждый из которых удерживает ресурс, ожидая другого.

Дедлок

```cpp
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx1, mtx2;

void task1() {
    std::lock_guard<std::mutex> lock1(mtx1);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock2(mtx2);
    std::cout << "Task 1 completed\n";
}

void task2() {
    std::lock_guard<std::mutex> lock2(mtx2);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock1(mtx1);
    std::cout << "Task 2 completed\n";
}

int main() {
    std::thread t1(task1);
    std::thread t2(task2);

    t1.join();
    t2.join();

    return 0;
}
```

До следующей лекции!