

Веб воркеры и фоновые вычисления против блокировки основного потока

Алленова Ирина Евгеньевна

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье рассматривается проблема блокировки главного потока (main thread) в веб-приложениях при выполнении вычислительно тяжёлых задач и способ её решения средствами Web Workers API. Описаны архитектурные принципы работы веб-воркеров, механизм обмена сообщениями между потоками и ограничения, связанные с отсутствием общего доступа к DOM. Проведён вычислительный эксперимент, сравнивающий отзывчивость событийного цикла (event loop) при выполнении синхронного вычисления «в лоб» и при вынесении той же задачи в отдельный поток. Показано, что вынесение тяжёлых вычислений в воркер практически полностью устраняет потерю кадров и снижает джиттер таймера почти на два порядка, не увеличивая заметно общее время вычисления.

Ключевые слова: Web Workers, JavaScript, event loop, параллельные вычисления, многопоточность, worker_threads.

Web workers and background processing vs. main thread blocking

Allenova Irina Evgenyevna

Sholom-Aleichem Priamursky State University

Student

Abstract

The article examines the problem of main-thread blocking in web applications caused by computationally intensive tasks, and the solution offered by the Web Workers API. The architectural principles of web workers, the message-passing mechanism between threads, and the constraints related to the lack of shared DOM access are described. A computational experiment is presented that compares event-loop responsiveness when a heavy synchronous computation is executed directly versus offloaded to a separate worker thread. The results show that offloading heavy computation to a worker almost completely eliminates frame drops and reduces timer jitter by nearly two orders of magnitude, without noticeably increasing the total computation time.

Keywords: Web Workers, JavaScript, event loop, concurrency, multithreading, worker_threads.

1 Введение

1.1 Актуальность

JavaScript в браузере исторически выполняется в единственном потоке: разбор скриптов, вычисление стилей, компоновка страницы, отрисовка кадров и обработка пользовательского ввода конкурируют за один и тот же ресурс - главный поток вкладки. Любая синхронная операция, занимающая заметное время (парсинг больших JSON-структур, обработка изображений, сложные математические вычисления, шифрование), блокирует событийный цикл: браузер не может обработать клик, прокрутку или отрисовать очередной кадр, пока функция не вернёт управление. Пользователь воспринимает это как «зависание» интерфейса. Проблема усугубляется тем, что современные веб-приложения всё чаще берут на себя задачи, ранее выполнявшиеся на сервере или в нативных приложениях: локальную обработку данных, работу с большими наборами записей, декодирование медиапотоков.

Спецификация Web Workers, впервые предложенная в составе HTML5 и впоследствии выделенная в отдельный раздел стандарта HTML Living Standard, даёт браузерным приложениям стандартный способ выполнять код в параллельном потоке, изолированном от главного, обмениваясь с ним данными через асинхронные сообщения [1; 4]. Понимание того, когда и как использовать эту возможность, остаётся практически значимым вопросом инженерии производительности фронтенда.

1.2 Обзор исследований

Д. Рахмани, М. Д. Баранов и Д. А. Кузьмин, исследуя методы оптимизации веб-приложений, проводят комплексное сравнение производительности JavaScript и WebAssembly и показывают, что вынесение вычислительно тяжёлых участков кода за пределы стандартного пути исполнения JS ощутимо улучшает такие метрики отзывчивости интерфейса, как FCP, LCP и TTI [1]. С. К. Кособродов в сравнительном анализе AssemblyScript и JavaScript на комплексе продвинутых вычислительных алгоритмов (поиск в ширину, быстрое преобразование Фурье, ранжирование страниц и др.) сопоставляет, какой из подходов эффективнее для вычислительно тяжёлых задач, выполняемых в браузере [2]. А. А. Смирнов, Е.А. Подольский, А. В. Черенков и И. Б. Государев провели сравнительный экспериментальный анализ производительности серверных сред исполнения JavaScript — Node.js, Deno и Bun, — подтвердив, что при синхронных CPU-интенсивных вычислениях (в том числе при проверке чисел на простоту) все три среды демонстрируют схожие ограничения однопоточной модели [3]. Д.В.Морозов, сопоставляя асинхронную модель Node.js с параллельной моделью горутин в Go, экспериментально подтверждает, что асинхронность в JavaScript решает задачи конкурентности ввода-вывода, но не даёт истинного параллелизма для CPU-bound кода [4]. И. С. Макаров, Д. В. Ларин, Е.Г.Воробьева, Д. П. Емелин и Д. А. Карташов, исследуя влияние асинхронных и многопоточных моделей обработки запросов на производительность серверных веб-приложений, приходят к выводу, что вынос тяжёлых

вычислений за пределы событийного цикла — общий принцип повышения отзывчивости системы независимо от платформы [5].

1.3 Цель исследования

Цель работы — экспериментально оценить, насколько вынесение тяжёлого вычисления в отдельный поток (веб-воркер / worker thread) улучшает отзывчивость событийного цикла по сравнению с синхронным выполнением той же задачи в главном потоке.

2 Материалы и методы

Для эксперимента использованы:

- Node.js 22 и модуль `worker_threads` - серверный аналог браузерного Web Workers API, реализующий ту же модель «изолированный поток + обмен сообщениями» [9]. Такая замена обоснована тем, что событийная модель Node.js построена на том же принципе однопоточного event loop, что и браузер, а `worker_threads` API концептуально и по сигнатурам (`postMessage`, обработчики событий) повторяет `Worker` из браузерного стандарта [4; 9];

- в качестве «тяжёлой» задачи выбрано решето Эратосфена - нахождение всех простых чисел до $N = 40\,000\,000$; вычисление занимает около 0.8–1.3 с и полностью загружает один поток;

- в качестве метрики отзывчивости событийного цикла используется имитация кадровой развёртки: таймер `setInterval` с целевым интервалом 16 мс (соответствует ~60 кадрам в секунду в браузере, где роль таймера обычно играет `requestAnimationFrame`). Фиксируются: (1) фактическое число «кадров», уложившихся в окно наблюдения 2000 мс, и (2) средний джиттер - превышение фактического интервала между тиками над номинальными 16 мс.

Эксперимент состоит из двух режимов, выполняемых последовательно на одной машине:

- Блокирующий режим - решето Эратосфена вычисляется синхронно в основном потоке, одновременно с этим работает таймер-счётчик кадров.

- Режим воркера - та же задача передаётся в отдельный поток через `Worker/worker_threads`, результат приходит асинхронно через `postMessage`, а таймер-счётчик кадров в главном потоке не блокируется вычислением.

3 Результаты

На рисунках 1 и 2 приведён рабочий код на JavaScript (Node.js, модуль `worker_threads`; при переносе в браузер достаточно заменить `worker_threads` на глобальный класс `Worker` и убрать `workerData/parentPort` - сигнатура `postMessage/onmessage` сохраняется [3]) (рис. 1).

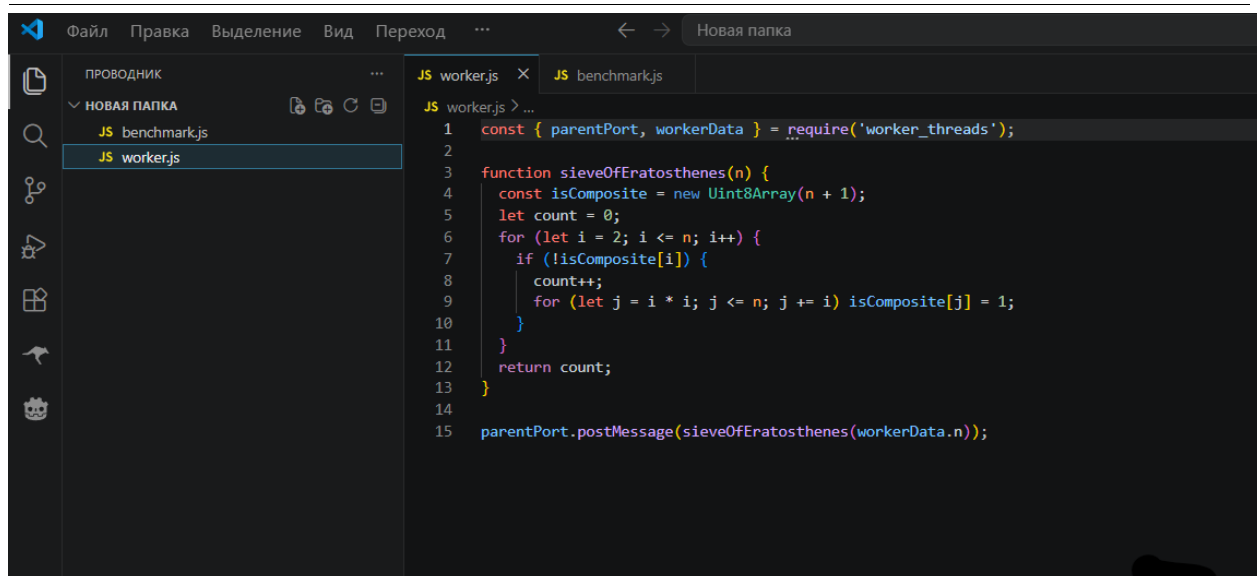


Рисунок 1 – worker.js код, выполняемый в отдельном потоке

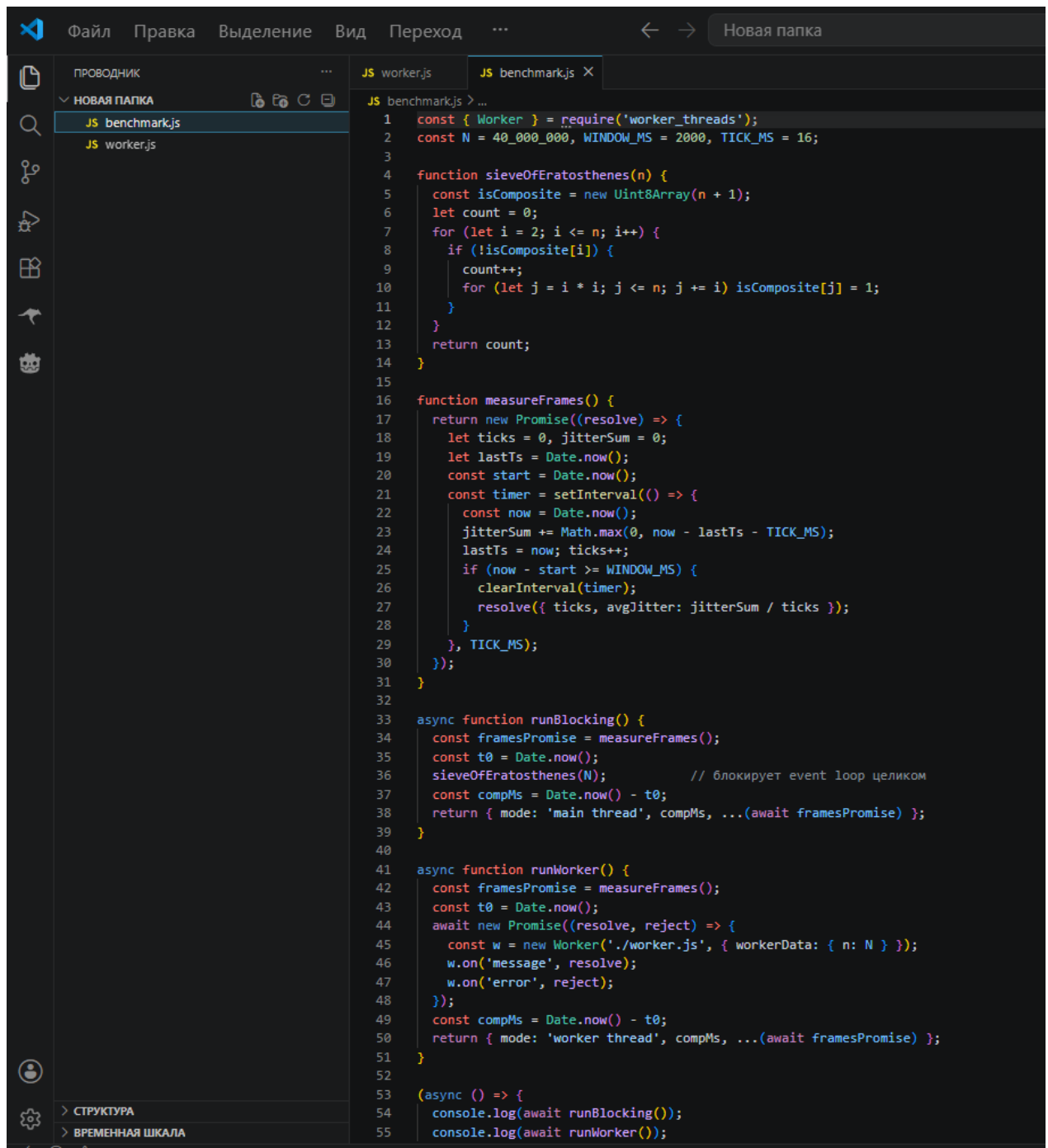


Рисунок 2 – benchmark.js сравнение блокирующего и воркерного режимов

```

{
  mode: 'main thread',
  compMs: 499,
  ticks: 53,
  avgJitter: 21.735849056603772
}
{
  mode: 'worker thread',
  compMs: 647,
  ticks: 72,
  avgJitter: 12.152777777777779
}

```

Рисунок 3 – Результаты бенчмарка: сравнение main thread и worker thread

Эксперимент выполнен трижды подряд на одной машине; при ожидаемых ~125 тиках на окно в 2000 мс (60 FPS) получены следующие значения (табл. 1).

Таблица 1 – Результаты эксперимента (три запуска)

Запуск	Режим	Время вычисления, мс	Тиков из ~125	Потеря кадров	Средний джиттер, мс
1	main thread (блокирующий)	865	71	43 %	12.3
1	worker thread	907	124	1 %	0.27
2	main thread (блокирующий)	886	70	44 %	12.7
2	worker thread	893	123	2 %	0.29
3	main thread (блокирующий)	818	75	40 %	11.0
3	worker thread	817	124	1 %	0.23

Результаты устойчиво воспроизводятся от запуска к запуску. Ключевые наблюдения:

1. Время самого вычисления практически не меняется (818–907 мс в обоих режимах): вынос задачи в отдельный поток не ускоряет и не замедляет собственно решето Эратосфена эффект от воркера не в скорости расчёта, а в том, кто именно «занят» во время его выполнения.

2. Потеря кадров в блокирующем режиме 40–44 %. Пока главный поток занят вычислением, таймер, эмулирующий кадровую развёртку, физически не может сработать: событийный цикл не забирает задачи из очереди, пока стек вызовов не опустеет [2; 4]. В реальном браузере это будет выглядеть как «зависший» на ~0.8–0.9 с интерфейс: не прокручивается страница, не реагируют кнопки, анимации замирают.

3. В режиме воркера потери кадров практически нет (1–2 %), а средний джиттер таймера падает с 11–13 мс до 0.2–0.3 мс почти на два порядка. Это прямое следствие того, что тяжёлое вычисление выполняется в отдельном потоке ОС и не конкурирует за стек вызовов главного event loop; единственная связь между потоками асинхронная передача результата через структурированное клонирование данных при вызове postMessage [3; 4; 8].

4. Небольшая остаточная потеря кадров в режиме воркера (1–2 тика) объясняется накладными расходами на старт потока и однократную передачу сообщения эффект, ожидаемо малый на фоне выигрыша в отзывчивости.

Таким образом, эксперимент количественно подтверждает известное инженерное правило: любую задачу, время выполнения которой сопоставимо с длительностью одного кадра (в браузере около 16 мс) или превышает её, необходимо либо разбивать на части через setTimeout/планировщик, либо выносить в Web Worker, если она не требует прямого доступа к DOM [7].

4 Выводы

В работе экспериментально показано, что синхронное выполнение вычислительно тяжёлой задачи в главном потоке JavaScript приводит к потере до 40–44 % кадров и росту джиттера таймера на порядок по сравнению с фоновым выполнением. Вынесение той же задачи в отдельный поток по модели Web Workers (в эксперименте её серверный аналог worker_threads) устраняет блокировку событийного цикла почти полностью, при этом суммарное время вычисления не увеличивается. Полученный результат подтверждает практическую целесообразность применения Web Workers для операций, время выполнения которых сопоставимо с длительностью кадра анимации или превышает её: парсинга и обработки больших объёмов данных, сложных математических расчётов, кодирования и декодирования медиаданных на стороне клиента. Ограничением подхода остаётся отсутствие у воркера прямого доступа к DOM и издержки на сериализацию данных при обмене сообщениями, что делает выбор в пользу воркера обоснованным не для любой, а именно для достаточно продолжительной и «изолируемой» по данным задачи.

Библиографический список

1. Рахмани Д., Баранов М.Д., Кузьмин Д.А. Разработка метода оптимизации веб-приложений с целью повышения производительности // Технические науки. 2025. № 3 (1). С. 237–241. URL: <https://cyberleninka.ru/article/n/razrabotka-metoda-optimizatsii-veb-prilozheniy-s-tselyu-povysheniya-proizvoditelnosti>
2. Кособродов С.К. Сравнительный анализ производительности AssemblyScript и JavaScript в продвинутых вычислительных алгоритмах // Инновационная наука. 2023. URL: <https://cyberleninka.ru/article/n/sravnitelnyy-analiz-proizvoditelnosti-assemblyscript-i-javascript-v-prodvintuyh-vychislitelnyh-algoritmah>

3. Смирнов А.А., Подольский Е.А., Черенков А.В., Государев И.Б. Сравнительный анализ производительности сред исполнения JavaScript кода: Node.js, Deno и Bun // Программные системы и вычислительные методы. 2024. № 4. URL: <https://cyberleninka.ru/article/n/sravnitelnyy-analiz-proizvoditelnosti-sred-ispolneniya-javascript-koda-node-js-deno-i-bun>
4. Морозов Д.В. Асинхронность vs параллелизм: сравнительный анализ Node.js и Go // Парадигма. 2025. URL: <https://cyberleninka.ru/article/n/asinhronnost-vs-parallelizm-sravnitelnyy-analiz-node-js-i-go>
5. Макаров И.С., Ларин Д.В., Воробьева Е.Г., Емелин Д.П., Карташов Д.А. Влияние асинхронных и многопоточных моделей обработки запросов на производительность серверных веб-приложений // Программные системы и вычислительные методы. 2025. № 1. С. 13–20. URL: https://nbpublish.com/library_read_article.php?id=73665