بسمه تعالی

پروژه دوم درس سیستم‌های عامل

پیاده‌سازی http server
با پشتیبانی از multithreading

روزبه معینی 401106544

دانشگاه صنعتی شریف

پاییز 1404

فهرست

۱. مقدمه

هدف از این گزارش، مستندسازی مراحل انجام‌شده در پروژه دوم درس سیستم‌های عامل است. در این تمرین به پیاده‌سازی سرور http و مقایسه حالت‌های مختلف پیاده‌سازی آن با استفاده از یک ریسه تا 16 ریسه است. همچنین به بررسی چالش‌های مختلف این پیاده‌سازی اعم از deadlock، busy waiting و ... پرداخته‌ایم. در نهایت کش LRU به سیستم اضافه و تاثیر آن‌را نیز بررسی کردیم.

نکته مهم: کدها، اسکریپت‌ها و نمودارها داخل سورس کامل قرار گرفته‌اند. فایل‌های لاگ حذف شدند تا پروژه مرتب تر تحویل شود ولی همگی با اجرای اسکریپت‌های متناظر که توضیحاتش در ادامه آمده‌است، قابل بازسازی هستند. در ادامه این بخش نحوه اجرای بخش‌های مختلف آمده‌است.

پیش‌نیازهای اجرا

برای کامپایل و اجرای صحیح این پروژه در محیط لینوکس (مانند Ubuntu)، ابزارهای زیر مورد نیاز است:

- کامپایلر G++ و make: برای کامپایل کدهای C++ با پشتیبانی از استاندارد C++11 یا بالاتر.
- کتابخانه pthread: این کتابخانه به صورت پیش‌فرض در توزیع‌های لینوکس موجود است و برای مدیریت ریسه‌های POSIX استفاده می‌شود.
- ابزار Apache Bench: برای انجام تست‌های بارگذاری و سنجش کارایی سرور.
- Valgrind: جهت بررسی نشت حافظه و مدیریت صحیح منابع سیستم.
- پایتون ۳ و کتابخانه Matplotlib: برای اجرای اسکریپت‌های تحلیل نتایج و رسم نمودارهای کارایی

اجزاء پروژه و نحوه اجرای آنها

کامپایل با دستور ساده make صورت میگیرد. اجرای سرور نیز به نحو زیر است:

```
./webserver -p [Port] -t [Threads] -r [Root_Directory] -c [Cache_Size_Bytes] [--noLRU]
```

۲. ساختار کد

در دو بخش به ساختار کد می‌پردازیم. یک بخش اصلی سورس که شامل کدهای cpp شامل کلاس‌های اصلی سرور است و فولدر www شامل html استاتیک سایت است. بخش دوم به اسکریپت‌های کمکی و خاص منظوره که برای رسم نمودارها و ترکیب‌های مختلف اجرا اضافه شده‌است می‌پردازیم.

بخش اول: پیاده‌سازی سرور

ساختار اصلی شامل 6 فایل پایتون است که شامل 6 کلاس اصلی سیستم است. این 6 کلاس به صورت abstract در یک مجموعه فایل هدر با اسم مشابه داخل فولدر include قرار گرفته‌است. این تعریف abstract کمک می‌کند هر کد فقط به هدر خودش رفرنس بزند و به جداسازی منطق بخش‌ها از هم کمک می‌کند.

 # کد main

تابع main نقطه ورود برنامه است که وظیفه مدیریت آرگومان‌های خط فرمان (مانند پورت و تعداد ریسه‌ها) و راه‌اندازی اجزای اصلی را بر عهده دارد. همچنین، مکانیزم مدیریت سیگنال (SIGINT) در این بخش پیاده‌سازی شده تا در صورت دریافت دستور توقف (Ctrl+C)، سرور به آرامی (Graceful Shutdown) بسته شده و تمام منابع آزاد شوند. تابع parsArgs در تصویر زیر امکان فراهم کردن فلگ‌های ممکن ورودی برای اجرای سرور را فراهم می‌کند تا سرور بتواند با ستاپ‌های مختلف طبق ساختار زیر اجرا شود. با فلگ noLRU میتوان کش را خاموش کرد.

webserver [-p port] [-t threads] [-r root_dir] [-c cache_bytes] [--noLRU]

```cpp
Config parseArgs(int argc, char* argv[]) {
    Config cfg;
    for (int i = 1; i < argc; ++i) {
        std::string arg = argv[i];
        if (arg == "-p" && i + 1 < argc) {
            cfg.port = std::atoi(argv[++i]);
        } else if (arg == "-t" && i + 1 < argc) {
            cfg.threads = static_cast<size_t>(std::atoi(argv[++i]));
        } else if (arg == "-r" && i + 1 < argc) {
            cfg.root = argv[++i];
        } else if (arg == "-c" && i + 1 < argc) {
            cfg.cache_bytes = static_cast<size_t>(std::atoll(argv[++i]));
        } else if (arg == "--noLRU" || arg == "--no-cache") {
            cfg.use_cache = false;
        } else if (arg == "-h" || arg == "--help") {
            std::cout << "Usage: webserver [-p port] [-t threads] [-r root_dir] [-c cache_bytes] [--noLRU]\n";
            std::exit(0);
        }
    }
    return cfg;
}

int main(int argc, char* argv[]) {
    Config cfg = parseArgs(argc, argv);

    Cache cache(cfg.cache_bytes);
    Stats stats;
    Cache* cache_ptr = cfg.use_cache ? &cache : nullptr;
    HttpHandler handler(cfg.root, cache_ptr, &stats);

    ThreadPool pool(cfg.threads, [&](int fd) { handler.handleClient(fd); });
    Server server(cfg.port, cfg.threads, cfg.root, &pool);
    g_server = &server;

    signal(SIGINT, handleSignal);
    signal(SIGTERM, handleSignal);

    if (!server.start()) {
        std::cerr << "Failed to start server" << std::endl;
        return 1;
    }

    std::cout << "Total requests handled: " << stats.totalRequests() << std::endl;
    std::cout << "Average response time (ms): " << stats.averageResponseMs() << std::endl;
    std::cout << "Cache hit rate: " << stats.cacheHitRate() * 100.0 << "%" << std::endl;
    std::cout << "Requests per second: " << stats.requestsPerSecond() << std::endl;

    return 0;
}
```

کد http_handler

این کلاس وظیفه Parse درخواست‌های خام HTTP و تولید پاسخ مناسب را بر عهده دارد. هندلر با بررسی متد درخواست GET و مسیر فایل، هدرهای مناسب را تنظیم می‌کند. همچنین مدیریت کدهای وضعیت مانند 200 OK، 404 Not Found و 400 Bad Request در این بخش انجام می‌شود.

در تصویر زیر خواندن از ورودی و تعریف تابع کمکی send and close که پاسخ را ارسال و اتصال را می‌بندد می‌بینیم.

```cpp
HttpHandler::HttpHandler(std::string root_dir, Cache* cache, Stats* stats)
    : root_dir_(std::move(root_dir)), cache_(cache), stats_(stats) {
    if (!root_dir_.empty() && root_dir_.back() == '/') {
        root_dir_.pop_back();
    }
}

void HttpHandler::handleClient(int client_fd) {
    auto start = std::chrono::steady_clock::now();
    bool cache_hit = false;

    std::string request;
    char buffer[4096];
    while (true) {
        ssize_t n = recv(client_fd, buffer, sizeof(buffer), 0);
        if (n <= 0) {
            break;
        }
        request.append(buffer, n);
        if (request.find("\r\n\r\n") != std::string::npos || request.size() > 8192) {
            break;
        }
    }

    auto sendAll = [](int fd, const std::string& data) {
        size_t total = 0;
        while (total < data.size()) {
            ssize_t sent = send(fd, data.data() + total, data.size() - total, 0);
            if (sent < 0) {
                if (errno == EINTR) {
                    continue;
                }
                break;
            }
            total += static_cast<size_t>(sent);
        }
    };
```

در تصویر بعدی در ادامه کد بالا بخش ارور هندلینگ را داریم:

```
if (request.empty() || request.size() > 8192) {
    sendAndClose(buildResponse(400, "text/plain", "Bad Request"));
    return;
}

size_t line_end = request.find("\r\n");
if (line_end == std::string::npos) {
    sendAndClose(buildResponse(400, "text/plain", "Bad Request"));
    return;
}

std::istringstream iss(request.substr(0, line_end));
std::string method, path, version;
iss >> method >> path >> version;

if (method.empty() || path.empty() || version.empty() || iss.fail()) {
    sendAndClose(buildResponse(400, "text/plain", "Bad Request"));
    return;
}

if (method != "GET") {
    sendAndClose(buildResponse(400, "text/plain", "Only GET supported"));
    return;
}

if (path.compare(0, 7, "http://") == 0 || path.compare(0, 8, "https://") == 0) {
    size_t scheme_end = path.find("//");
    size_t after_host = (scheme_end == std::string::npos) ? std::string::npos : path.find('/', scheme_end + 2)
    path = (after_host == std::string::npos) ? "/" : path.substr(after_host);
}

size_t query_pos = path.find('?');
if (query_pos != std::string::npos) {
    path = path.substr(0, query_pos);
}
size_t frag_pos = path.find('#');
if (frag_pos != std::string::npos) {
    path = path.substr(0, frag_pos);
}

if (path.empty() || path[0] != '/' || !isPathSafe(path)) {
    sendAndClose(buildResponse(400, "text/plain", "Bad Request"));
    return;
}
```

چک کردن کش در ادامه کد بالا با استفاده از کلاس کش صورت می‌پذیرد که در بعد بیشتر به پیاده‌سازی آن می‌پردازیم:

```
if (cache_ && cache_->get(full_path, body)) {
    cache_hit = true;
} else {
    if (!readFile(full_path, body)) {
        sendAndClose(buildResponse(404, "text/plain", "Not Found"));
        if (stats_) {
            stats_->recordRequest(std::chrono::steady_clock::now() - start, false);
        }
        return;
    }
    if (cache_) {
        cache_->put(full_path, body);
    }
}

std::string response = buildResponse(200, guessContentType(full_path), body);
sendAndClose(response);

if (stats_) {
    stats_->recordRequest(std::chrono::steady_clock::now() - start, cache_hit);
}
```

 # کد کش

برای بهبود کارایی، یک کش با سیاست Least Recently Used پیاده‌سازی شده است. این کش فایل‌های پرکاربرد را در حافظه رم نگه می‌دارد تا از عملیات ورودی/خروجی دیسک جلوگیری کند. ساختار داده شامل یک HashMap برای دسترسی سریع و یک لیست پیوندی دوطرفه برای مدیریت سیاست حذف استفاده است. دسترسی به کش نیز توسط Mutex محافظت می‌شود.

توابع آن عبارتند از:

- get: عملیات thread safe با استفاده از lock_guard. در صورت موجود بودن فایل، تابع touch برای بروزرسانی اولویت فراخوانی شده و شمارنده hits افزایش می‌یابد؛ در غیر این صورت misses ثبت می‌شود.

```
bool Cache::get(const std::string& key, std::string& value) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = map_.find(key);
    if (it == map_.end()) {
        ++misses_;
        return false;
    }

    touch(key, it->second.second);
    value = it->second.first;
    ++hits_;
    return true;
}
```

- put: درج یا جایگزینی محتوا. پس از افزودن داده، حجم فعلی آپدیت می‌شود و تابع evictIfNeeded بررسی می‌کند که آیا نیاز به آزادسازی فضا وجود دارد یا خیر.

```
void Cache::put(const std::string& key, const std::string& value) {
    std::lock_guard<std::mutex> lock(mutex_);
    auto it = map_.find(key);
    if (it != map_.end()) {
        current_bytes_ -= it->second.first.size();
        it->second.first = value;
        touch(key, it->second.second);
    } else {
        lru_.push_front(key);
        map_[key] = {value, lru_.begin()};
    }
    current_bytes_ += value.size();
    evictIfNeeded();
}
```

- touch: وظیفه حفظ ترتیب زمانی را دارد؛ این تابع نود فایل موردنظر را به ابتدای لیست پیوندی منتقل می‌کند تا به عنوان اخیرا استفاده شده علامت‌گذاری شود.

- evictIfNeeded: سیاست اخراج (Eviction) را اجرا می‌کند. تا زمانی که حجم اشغال شده بیشتر از حد مجاز باشد، داده‌های انتهای لیست یعنی قدیمی‌ترها را حذف می‌کند.

```
void Cache::evictIfNeeded() {
    while (current_bytes_ > max_bytes_ && !lru_.empty()) {
        std::string victim = lru_.back();
        lru_.pop_back();
        auto it = map_.find(victim);
        if (it != map_.end()) {
            current_bytes_ -= it->second.first.size();
            map_.erase(it);
        }
    }
}
```

- hitRate و sizeBytes: توابع کمکی که وضعیت لحظه‌ای سیستم را برمی‌گردانند؛ hitRate نسبت موفقیت کش را محاسبه کرده و sizeBytes میزان حافظه درگیر را گزارش می‌دهد.

 # کد استخر ریسه

استخر ریسه قلب پردازش هم‌زمان سرور است. این کلاس تعدادی ریسه کارگر را هنگام شروع برنامه ایجاد می‌کند که همگی منتظر دریافت وظیفه از یک صف مشترک هستند. برای جلوگیری از شرایط رقابتی از mutex جهت قفل کردن صف و برای جلوگیری از busy waiting از condition variable استفاده شده است. مهم‌ترین تابع این کد، تابع کارگر است:

```
void* ThreadPool::workerTrampoline(void* arg) {
    ThreadPool* pool = static_cast<ThreadPool*>(arg);
    pool->workerLoop();
    return nullptr;
}

void ThreadPool::workerLoop() {
    while (true) {
        int client_fd = -1;

        pthread_mutex_lock(&mutex_);
        while (tasks_.empty() && !stopping_) {
            pthread_cond_wait(&cond_, &mutex_);
        }
        if (stopping_ && tasks_.empty()) {
            pthread_mutex_unlock(&mutex_);
            break;
        }
        client_fd = tasks_.front();
        tasks_.pop();
        pthread_mutex_unlock(&mutex_);

        if (client_fd != -1) {
            handler_(client_fd);
        }
    }
```

سه تابع مهم دیگر نیز عبارتند از:

ThreadPool: سازنده کلاس که ریسه‌های کارگر را ایجاد و متغیرهای همگام‌سازی را مقداردهی می‌کند.

enqueue: این تابع توسط ریسه اصلی فراخوانی می‌شود تا یک اتصال جدید را در صف قرار دهد. پس از افزودن وظیفه، با استفاده از pthread_cond_signal یکی از ریسه‌های خوابیده را بیدار می‌کند.

```
void ThreadPool::enqueue(int client_fd) {
    pthread_mutex_lock(&mutex_);
    if (stopping_) {
        pthread_mutex_unlock(&mutex_);
        close(client_fd);
        return;
    }
    tasks_.push(client_fd);
    pthread_cond_signal(&cond_);
    pthread_mutex_unlock(&mutex_);
}
```

shutdown: وظیفه خاتمه امن سرور را بر عهده دارد. این تابع با broadcast کردن سیگنال، تمام ریسه‌ها را بیدار کرده و با pthread_join منتظر پایان کار آن‌ها می‌ماند

```
void ThreadPool::shutdown() {
    pthread_mutex_lock(&mutex_);
    if (stopping_) {
        pthread_mutex_unlock(&mutex_);
        return;
    }
    stopping_ = true;
    pthread_cond_broadcast(&cond_);
    pthread_mutex_unlock(&mutex_);

    for (pthread_t& t : threads_) {
        if (t) {
            pthread_join(t, nullptr);
        }
    }
}
```

بر اساس کد پیاده‌سازی شده، می‌توان ثابت کرد که سیستم دچار مشکلات زیر نمی‌شود:

۱. جلوگیری از Busy Waiting: در تابع workerLoop از دستور pthread_cond_wait استفاده شده است. این دستور باعث می‌شود ریسه در صورتی که صف خالی باشد (tasks_.empty)، توسط سیستم‌عامل به حالت خواب برود و هیچ چرخه پردازشی از CPU مصرف نکند. ریسه تنها زمانی بیدار می‌شود که سیگنالی مبنی بر اضافه شدن کار جدید دریافت کند.

۲. جلوگیری از بن‌بست: کد از یک الگوی ساده و یک‌نواخت برای قفل‌گذاری استفاده می‌کند:

- ترتیب ثابت: تمام توابع یعنی enqueue, shutdown, workerLoop ابتدا mutex_ را قفل کرده و پس از اتمام کار با صف، آن را آزاد می‌کنند. هیچ‌گاه یک ریسه سعی نمی‌کند در حالی که یک قفل را در اختیار دارد، قفل دیگری را تصاحب کند.

- مدیریت انتظار: تابع pthread_cond_wait به صورت اتمیک قفل را آزاد می‌کند تا سایر ریسه‌ها یا تولیدکننده بتوانند به صف دسترسی داشته باشند، که این امر مانع از مسدود شدن دائمی سیستم می‌شود.

 # کد سرور و stats

کلاس سرور مسئولیت مدیریت سوکت‌ها و ارتباطات شبکه را بر عهده دارد. این بخش سوکت اصلی را روی پورت مشخص شده bind و listen می‌کند. حلقه اصلی پذیرش در اینجا قرار دارد که اتصال کلاینت را دریافت کرده و فایل دیسکریپتور (fd) آن را به عنوان یک وظیفه به استخر ریسه می‌سپارد تا بلافاصله برای درخواست بعدی آماده شود. راه‌اندازی سوکت:

```cpp
bool Server::setupSocket() {
    listen_fd_ = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd_ < 0) {
        std::perror("socket");
        return false;
    }
    int opt = 1;
    if (setsockopt(listen_fd_, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0) {
        std::perror("setsockopt");
        return false;
    }
    sockaddr_in addr{};
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(port_);
    if (bind(listen_fd_, reinterpret_cast<sockaddr*>(&addr), sizeof(addr)) < 0) {
        std::perror("bind");
        return false;
    }
    if (listen(listen_fd_, SOMAXCONN) < 0) {
        std::perror("listen");
        return false;
    }
    return true;
}
```

در نهایت کد محاسبه‌گر پارامترها را داریم که برای بررسی پارامترهای سیستم استفاده می‌شود.

```cpp
Stats::Stats()
    : total_requests_(0), cache_hits_(0), start_time_(std::chrono::steady_clock::now()), total_latency_(0) {}

void Stats::recordRequest(std::chrono::steady_clock::duration duration, bool cache_hit) {
    std::lock_guard<std::mutex> lock(mutex_);
    ++total_requests_;
    if (cache_hit) {
        ++cache_hits_;
    }
    total_latency_ += std::chrono::duration_cast<std::chrono::nanoseconds>(duration);
}

double Stats::requestsPerSecond() const {
    std::lock_guard<std::mutex> lock(mutex_);
    auto elapsed = std::chrono::steady_clock::now() - start_time_;
    double seconds = std::chrono::duration_cast<std::chrono::duration<double>>(elapsed).count();
    if (seconds <= 0.0) {
        return 0.0;
    }
    return static_cast<double>(total_requests_) / seconds;
}

double Stats::averageResponseMs() const {
    std::lock_guard<std::mutex> lock(mutex_);
    if (total_requests_ == 0) {
        return 0.0;
    }
    double avg_ns = static_cast<double>(total_latency_.count()) / static_cast<double>(total_requests_);
    return avg_ns / 1'000'000.0;
}

double Stats::cacheHitRate() const {
    std::lock_guard<std::mutex> lock(mutex_);
    if (total_requests_ == 0) {
        return 0.0;
    }
    return static_cast<double>(cache_hits_) / static_cast<double>(total_requests_);
}

uint64_t Stats::totalRequests() const {
    std::lock_guard<std::mutex> lock(mutex_);
    return total_requests_;
}
```

بخش دوم: اسکریپت‌ها و کدهای کمکی برای اجرا

فایل makefile

برای کامپایل پروژه از یک Makefile استاندارد استفاده شده است که وابستگی‌های فایل‌های cpp و h را مدیریت می‌کند. این اسکریپت با فلگ -pthread لینک شدن صحیح کتابخانه ریسه‌ها را انجام می‌دهد. این اسکریپت ارزش علمی خاصی ندارد و صرفا از جزئیات پیاده‌سازی است و داخل سورس قرار دارد.

اسکریپت ترکیبی run_bench و کد پایتون رسم نمودار

جهت خودکارسازی فرآیند تست و ارزیابی، اسکریپتی توسعه داده شده که سرور را با پیکربندی‌های مختلف یعنی با تعداد ریسه متفاوت از 1 تا 16 اجرا کرده و با ab بر اساس کانفیگ ورودی تعداد و میزان همروندی کاربران آن را بنچمارک می‌کند.

نحوه استفاده از اسکریپت به شرح زیر است:

bash scripts/run_bench.sh [N] [C] [URL] [--auto|--noLRU|--cache-only]

N تعداد درخواست‌هاست و C درجه همروندی را نشان می‌دهد. URL آدرسی است که دستور ab روی آن اجرا می‌شود. سه حالت اجرا داریم که با پرچم مشخص می‌شوند. حالت پیشفرض، اتوماتیک هر دو حالت کش خاموش و روشن را اجرا می‌کند و نمودار ترکیبی تحویل می‌دهد. اگر صرفا به نتایج یکی از حالت‌های با یا بدون کش نیاز داشتیم با تنظیم پرچم اینکار را می‌کنیم.

تابع اجرا کننده به ازای تمام تعدادهای ریسه‌ها:

```
run_suite() {
  local cache_flag="$1"  # on/off
  local cache_label="${cache_flag}"
  local no_lru_arg=()
  if [[ "${cache_flag}" == "off" ]]; then
    no_lru_arg=("--noLRU")
  fi

  for T in ${THREADS}; do
    echo "Running threads=${T} n=${N} c=${C} url=${TARGET_URL} cache=${cache_label}"
    SERVER_ARGS=("-p" "${PORT}" "-t" "${T}" "-r" "${ROOT}" "${no_lru_arg[@]}")
    "${BINARY}" "${SERVER_ARGS[@]}" > "${LOG_DIR}/server_${T}_${cache_label}.log" 2>&1 &
    SERVER_PID=$!
    sleep 1

    ab_log="${LOG_DIR}/ab_${T}_${cache_label}.log"
    if ! ab -n "${N}" -c "${C}" "${TARGET_URL}" > "${ab_log}" 2>&1; then
      echo "ab failed for threads=${T} cache=${cache_label}, see ${ab_log}" >&2
      cleanup
      continue
    fi

    cleanup
    SERVER_PID=""

    rps=$(grep "Requests per second" "${ab_log}" | awk '{print $4}')
    tpr=$(grep -m1 "Time per request" "${ab_log}" | awk '{print $4}')
    # ab prints: "Transfer rate: 12345.67 [Kbytes/sec] received"; the 3rd field is the number.
    xfer=$(grep "Transfer rate" "${ab_log}" | awk '{print $3}')
    failed=$(grep "Failed requests" "${ab_log}" | awk '{print $3}')

    echo -e "${T}\t${N}\t${C}\t${cache_label}\t${rps:-0}\t${tpr:-0}\t${xfer:-0}\t${failed:-0}" >> "${results_t
    echo "Finished threads=${T} cache=${cache_label}: rps=${rps:-0}, time_per_req_ms=${tpr:-0}, transfer_kBps=
    sleep 1
  done
}
```

خروجی این تست‌ها پارس شده و توسط با کمک کتابخانه matplotlib پایتون و یک کد کوچک به نمودارهای مقایسه‌ای تبدیل می‌شود. این کد کارخاصی نمیکند صرفا خروجی ترمینالی بخش قبل را به نمودارهای قابل استفاده تبدیل می‌کند. برای مشاهده این دو کد می‌توان به فولدر scripts مراجعه کرد.

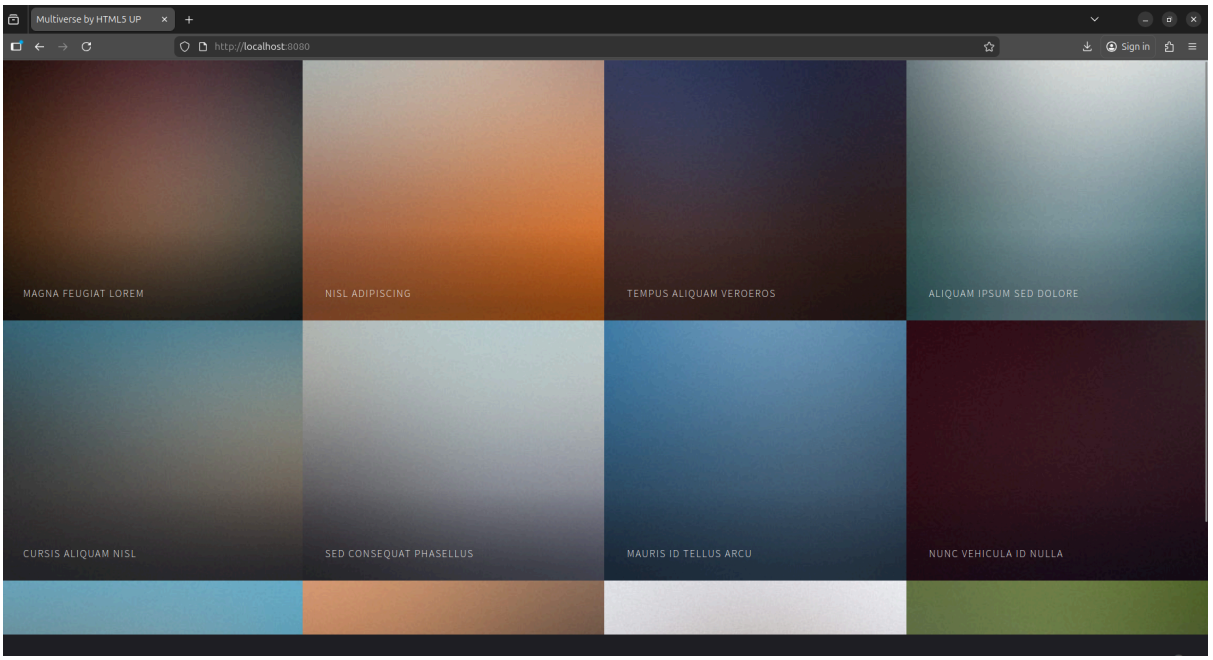
۳. نتیجه اجرا و مقایسه

در دو بخش به بررسی نتیجه اجرا می‌پردازیم. اول از سناریوهای ساده دستی شروع می‌کنیم. سپس سراغ اسکریپت‌ها، تحلیل و بررسی و مقایسه حالت‌های مختلف می‌پردازیم.

بخش اول: تست‌های دستی

 # درخواست ساده از ترمینال و مرورگر

پس از اجرای ساده سرور، داخل پورت 8080 میتوانیم ببینیم که سایت به خوبی بالا می‌آید. در فولدر www سورس mtliverse از خود گروه html up به منظور تست استفاده شده‌است.



نتیجه تک اجرا داخل ترمینال نیز با ابزار nc مشخصا بدون مشکل صورت می‌پذیرد. در این تست و تست‌های بعدی همیشه کش روشن است صرفا هنگام مقایسه تاثیر کش آن‌را خاموش می‌کنیم.

```
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/MutliThreadHttpServer$ echo -e "GET /index.html HTTP/1\r\n\r\n" | nc localhost 8080
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 7004
Connection: close

<!DOCTYPE HTML>
<!--
        Multiverse by HTML5 UP
        html5up.net | @ajlkn
        Free for personal and commercial use under the CCA 3.0 license (html5up.net/license)
-->
<html>
```

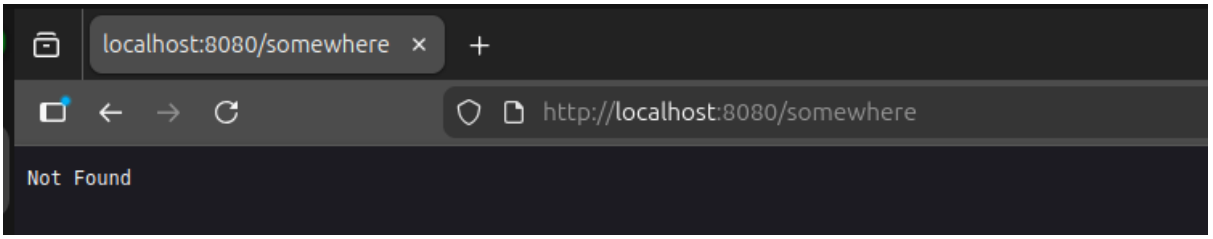
ارور 400 bad request

با استفاده از اسم متود بی ربط و nc درخواست اشتباه به سرور میدهیم تا عمکرد ارور 400 را طبق تصویر زیر بسنجیم.

```
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/Project2$ echo -e "DELETE_ME /index.html HTTP/1.1\r\n\r\n" | nc localhost 8080
HTTP/1.1 400 Bad Request
Content-Type: text/plain
Content-Length: 18
Connection: close
```

ارور 404 not found

با وارد کردن یک زیر آدرس بیربط داخل مرورگر، 404 به خوبی پاسخ داده می‌شود:



تست با ab و فایل باینری

برای تست بنچمارک اولیه با این ابزار ابتدا سرور را اجرا می‌کنیم سپس ab را با تعداد درخواست 10000 عدد و درجه همروندی 100 روی یک فایل عکس اجرا می‌کنیم:

```
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/MutliThreadHttpServer$ ab -n 10000 -c 100 http://localhost:8080/images/fulls/01.jpg
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /images/fulls/01.jpg
Document Length:        48762 bytes

Concurrency Level:      100
Time taken for tests:   0.751 seconds
Complete requests:      10000
Failed requests:        0
Total transferred:      488490000 bytes
HTML transferred:       487620000 bytes
Requests per second:    13313.51 [#/sec] (mean)
Time per request:       7.511 [ms] (mean)
Time per request:       0.075 [ms] (mean, across all concurrent requests)
Transfer rate:          635108.80 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    2   0.9      2       5
Processing:     0    5   1.4      5      13
Waiting:        0    1   0.8      1       8
Total:          3    7   1.4      7      15
```

 # مقایسه اولیه حالات مختلف

همینجا با استفاده از ab میتوانیم یک مقایسه اولیه بین حالت‌های مختلف بکنیم. برای اینکار درخواست ab مشابه برای فایل باینری را سه مرتبه با سه کانفیگ مختلف سرور همزمان اجرا کردیم.

```
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/Project2$ ab -n 10000 -c 100 http://localh
ost:8080/images/fulls/01.jpg
This is ApacheBench, Version 2.3 <$Revision: 1903618 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests
```

همانطور که در تصویر بعدی مشخص است در حالت اول 8 ریسه داریم و کش خاموش است. با روشن کردن کش در حالت دوم بدون تغییر تعداد ریسه، از میزان متوسط زمان پاسخگویی حدود 20.8 میلی ثانیه به 18.6 در حالت کش روشن رسیدیم. همچنین در تعداد درخواست‌های پردازش شده در ثانیه نیز افزایش حدودا 7 در خواست دیده می‌شود.

اما در اجرای سوم بدون خاموش کردن کش، اینبار تک ریسه برنامه را اجرا می‌کنیم. اینبار میزان بیش از 50 درصد کاهش تعداد درخواست در ثانیه به مقدار 115 را در خروجی مشاهده می‌کنیم. با اینحال میانگین زمان پاسخگویی در حالت تک ریسه بهتر است که این بخاطر کم حجم بودن فایل باینری و زیاد بودن سربار ریسه‌ها میتواند باشد. در ادامه خواهیم دید در فایل‌های سنگین تر و با ترافیک شلوغ تر، این میزان نیز در حالت چندریسه‌ای بهبود می‌یابد.

```
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/Project2$ ./webserver -p 8080 -t 8 -r www -c 5000000 --noLRU
Server listening on port 8080 (root: www)
^C
Shutting down...
Total requests handled: 10000
Average response time (ms): 20.8598
Cache hit rate: 0%
Requests per second: 255.728
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/Project2$ ./webserver -p 8080 -t 8 -r www -c 5000000
Server listening on port 8080 (root: www)
^C
Shutting down...
Total requests handled: 10000
Average response time (ms): 18.6739
Cache hit rate: 99.99%
Requests per second: 262.834
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/Project2$ ./webserver -p 8080 -t 1 -r www
-c 5000000
Server listening on port 8080 (root: www)
^C
Shutting down...
Total requests handled: 10000
Average response time (ms): 5.35705
Cache hit rate: 99.99%
Requests per second: 115.3
```

تست با valgrind

برای تضمین مدیریت صحیح حافظه، سرور تحت نظارت ابزار Valgrind تست شد. خروجی نشان می‌دهد که حتی پس از یک میلیون درخواست و خاتمه برنامه، هیچ‌گونهMemory Leak وجود ندارد. این نشان‌دهنده آزادسازی صحیح تمام منابع، بافرها و بستن فایل دیسکریپتورها است.

```
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/Project2$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
./webserver -p 8080 -t 8 -r www
==14862== Memcheck, a memory error detector
==14862== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==14862== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==14862== Command: ./webserver -p 8080 -t 8 -r www
==14862==
Server listening on port 8080 (root: www)
^C
Shutting down...
Total requests handled: 1000000
Average response time (ms): 1.02432
Cache hit rate: 99.9999%
Requests per second: 2333.49
==14862==
==14862== HEAP SUMMARY:
==14862==     in use at exit: 0 bytes in 0 blocks
==14862==   total heap usage: 9,007,835 allocs, 9,007,835 frees, 30,107,109,306 bytes allocated
==14862==
==14862== All heap blocks were freed -- no leaks are possible
==14862==
==14862== For lists of detected and suppressed errors, rerun with: -s
==14862== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/Project2$ ./webserver -p 8080 -t 8 -r www -c 5000000
```

بررسی صحت همروندی

یکبار اجرای دستور ab بدون مشکل با کانفیگ سنگین صحت همروندی را به صورت ضمنی تایید می‌کند. تمام این درخواست‌ها وارد صف می‌شوند و دسترسی به اطلاعات صف ذاتا یک منبع مشترک است. وقتی تمام این درخواست‌ها بدون مشکل وارد صف و پردازش می‌شوند یعنی سیستم کنترل همروندی ما درست کار می‌کند:

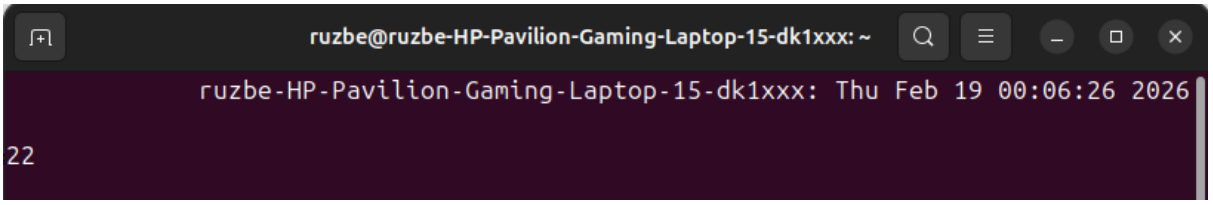
```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /images/fulls/01.jpg
Document Length:        48762 bytes

Concurrency Level:      1000
Time taken for tests:   67.804 seconds
Complete requests:      1000000
Failed requests:        0
Total transferred:      48849000000 bytes
HTML transferred:       48762000000 bytes
Requests per second:    14748.38 [#/sec] (mean)
Time per request:       67.804 [ms] (mean)
Time per request:       0.068 [ms] (mean, across all concurrent requests)
Transfer rate:          703558.06 [Kbytes/sec] received
```

 # مونیتور با htop و بررسی فایل‌های باز با lsof

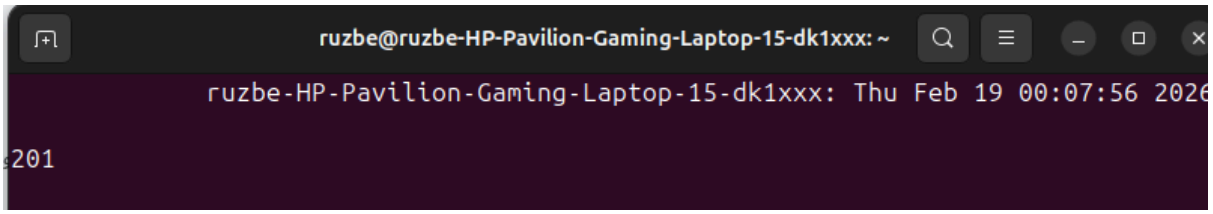
اگر همزمان با اجرای یک دستور ab طولانی، نگاهی به htop انداخته و با دیدن اطلاعات ریسه‌ها و یافتن شماره پردازه، دستور lsof را برای پردازه اجرا کنیم، مشاهده می‌کنیم فایل‌های باز به خوبی با افزایش لود زیاد می‌شوند و با موفقیت همه بعد از پایان بنچمارک به حالت قبلی برمیگردند و بسته می‌شوند. طبق تصویر ریز شماره پردازه 13634 است و مابقی ریسه هستند و رنگ سبز دارند.

```
15549 ruzbe       20   0 58180 25404  6348 R 105.0  0.2  0:12.50 ab -n 1000000 -c 1000 http://localhost:8080/
13638 ruzbe       20   0  582M  6088  3512 S  11.5  0.0  0:02.06 ./webserver -p 8080 -t 8 -r www -c 5000000
13640 ruzbe       20   0  582M  6088  3512 S  11.5  0.0  0:02.07 ./webserver -p 8080 -t 8 -r www -c 5000000
13635 ruzbe       20   0  582M  6088  3512 S  11.0  0.0  0:02.06 ./webserver -p 8080 -t 8 -r www -c 5000000
13636 ruzbe       20   0  582M  6088  3512 S  11.5  0.0  0:02.07 ./webserver -p 8080 -t 8 -r www -c 5000000
13637 ruzbe       20   0  582M  6088  3512 S  11.5  0.0  0:02.07 ./webserver -p 8080 -t 8 -r www -c 5000000
13641 ruzbe       20   0  582M  6088  3512 S  11.5  0.0  0:02.08 ./webserver -p 8080 -t 8 -r www -c 5000000
13642 ruzbe       20   0  582M  6088  3512 S  11.0  0.0  0:02.05 ./webserver -p 8080 -t 8 -r www -c 5000000
13639 ruzbe       20   0  582M  6088  3512 S  11.5  0.0  0:02.05 ./webserver -p 8080 -t 8 -r www -c 5000000
13634 ruzbe       20   0  582M  6088  3512 R   7.3  0.0  0:01.42 ./webserver -p 8080 -t 8 -r www -c 5000000
```

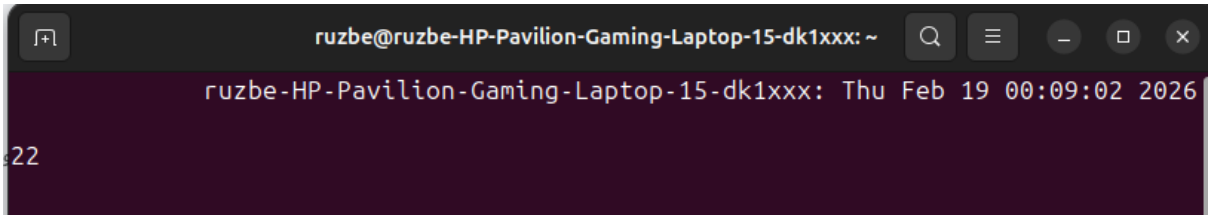
قبل از اجرای بنچمارک و سرور در حال اجرا:



هنگام افزایش بار: (عدد نوسان می‌کند اسکرین شات در یک لحظه است)



بعد از پایان بنچمارک:



به زمان هر تصویر برای صحت سنجی پشت سر هم بودنشان توجه کنید.

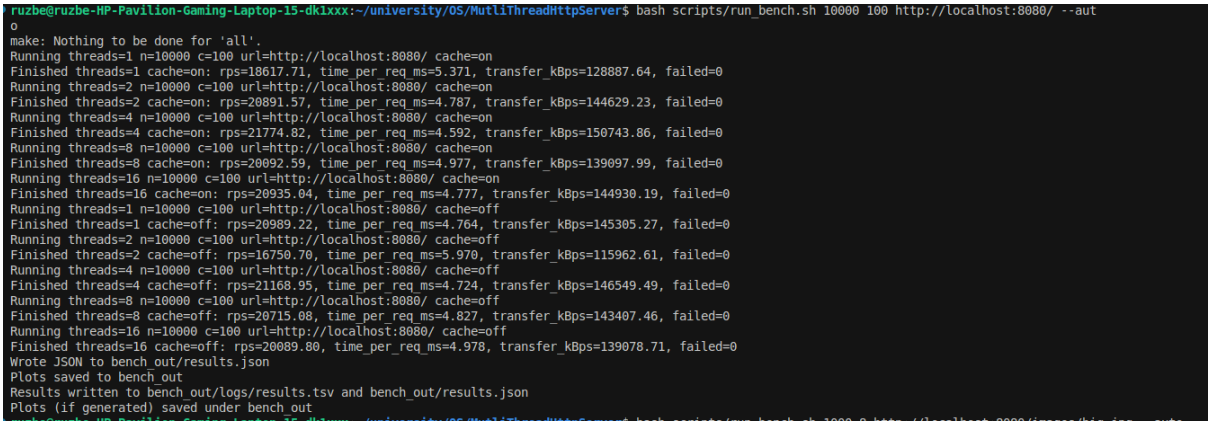
بخش دوم: تست‌های ترکیبی و تحلیل

در این بخش اسکریپت نوشته شده را با ترکیب فایل‌های مختلف از بار سبک تا سنگین تست کرده‌ایم تا تاثیرات کش و ریسه‌ها را دقیق‌تر بررسی کنیم.

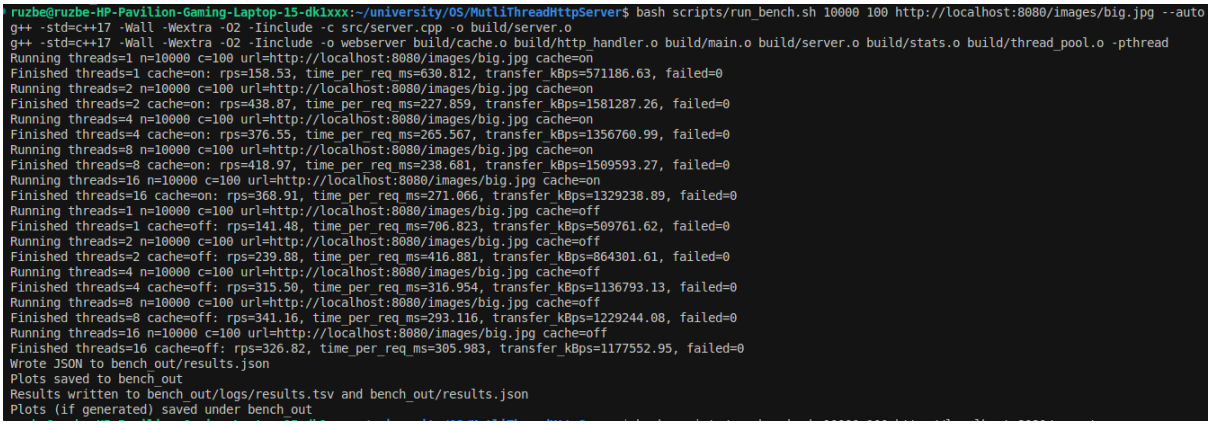
 # اجرای کانفیگ‌های مختلف اسکریپت ترکیبی

اجرای اول یک مرتبه با روشن بودن کش و برای صرف بررسی صرف تاثیر تغییر ریسه‌ها و صحت عملکرد فلگ انجام شده‌است.

در اجرای دوم لود سیستم را روی یک فایل html ساده که همان html اصلی است سنجیده‌ایم



در اجرای سوم لود را روی فایل‌های باینری سنگین مثل big.jgp برده‌ایم

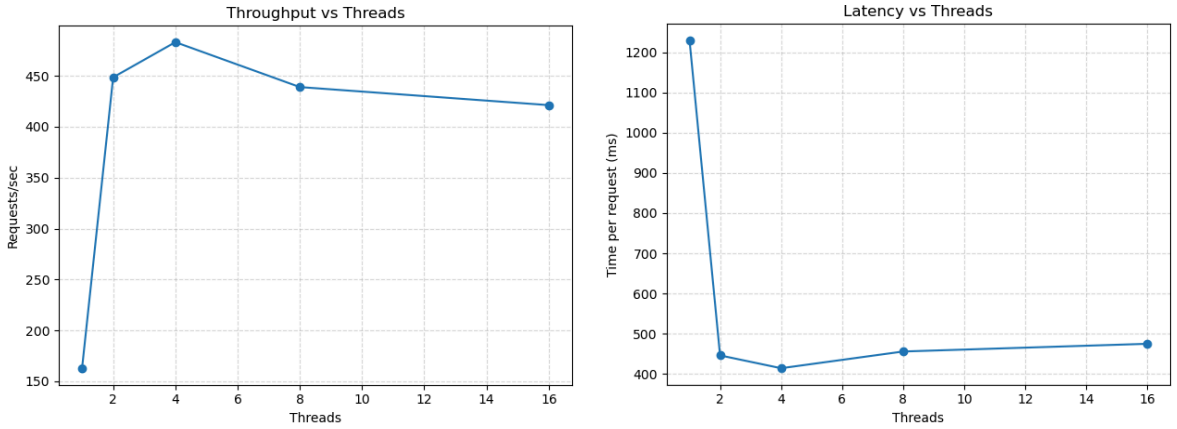


و در نهایت عملکرد سیستم را در یک سناریو با همروندی پایین و درخواست‌های کم نگاه کرده‌ایم:

```
ruzbe@ruzbe-HP-Pavilion-Gaming-Laptop-15-dk1xxx:~/university/OS/MutliThreadHttpServer$ bash scripts/run_bench.sh 1000 2 http://localhost:8080/images/bi
g.jpg --auto
make: Nothing to be done for 'all'.
Running threads=1 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=on
Finished threads=1 cache=on: rps=146.93, time_per_req_ms=13.612, transfer_kBps=529402.49, failed=0
Running threads=2 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=on
Finished threads=2 cache=on: rps=348.70, time_per_req_ms=5.736, transfer_kBps=1256407.30, failed=0
Running threads=4 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=on
Finished threads=4 cache=on: rps=346.90, time_per_req_ms=5.765, transfer_kBps=1249918.84, failed=0
Running threads=8 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=on
Finished threads=8 cache=on: rps=367.16, time_per_req_ms=5.447, transfer_kBps=1322931.46, failed=0
Running threads=16 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=on
Finished threads=16 cache=on: rps=364.11, time_per_req_ms=5.493, transfer_kBps=1311930.07, failed=0
Running threads=1 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=off
Finished threads=1 cache=off: rps=134.65, time_per_req_ms=14.853, transfer_kBps=485174.34, failed=0
Running threads=2 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=off
Finished threads=2 cache=off: rps=201.14, time_per_req_ms=9.943, transfer_kBps=724727.75, failed=0
Running threads=4 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=off
Finished threads=4 cache=off: rps=186.36, time_per_req_ms=10.732, transfer_kBps=671482.31, failed=0
Running threads=8 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=off
Finished threads=8 cache=off: rps=202.05, time_per_req_ms=9.899, transfer_kBps=727995.45, failed=0
Running threads=16 n=1000 c=2 url=http://localhost:8080/images/big.jpg cache=off
Finished threads=16 cache=off: rps=194.55, time_per_req_ms=10.280, transfer_kBps=700997.34, failed=0
Wrote JSON to bench_out/results.json
Plots saved to bench_out
Results written to bench_out/logs/results.tsv and bench_out/results.json
Plots (if generated) saved under bench_out
```

 # تحلیل‌های نمودارهای به دست‌آمده

نمودار تک حالته اول که صرفا برای بررسی تاثیر تعداد ریسه بود، خروجی زیر را می‌دهد:

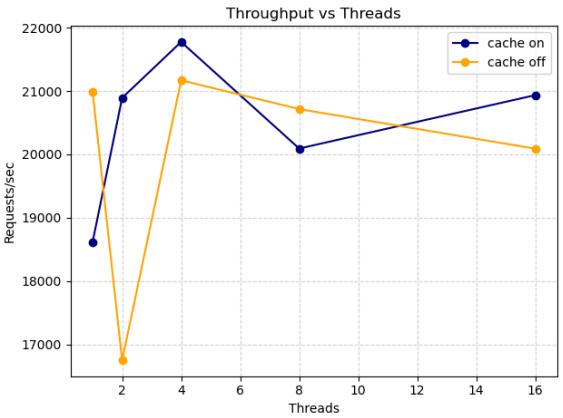


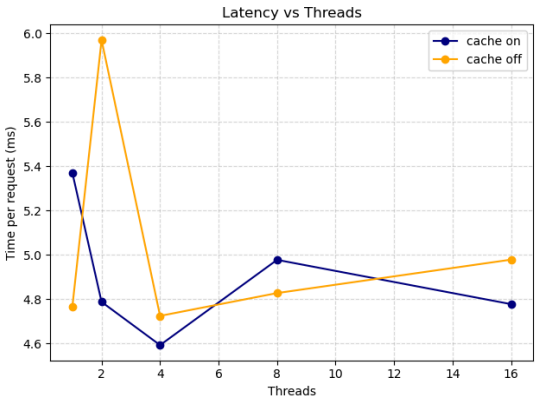
نتیجه نمودارهای بالا خیلی جالب است. پردازنده سیستم ما 4 هسته حقیقی و 8 ریسه سخت‌افزاری دارد. همانطور که در تصویر مشخص است، بهترین عملکرد از آن 4 ریسه شده‌است که دقیقا برابر با تعداد هسته‌های ماست. در این حالت بهترین تنظیم میان افزایش همروندی بدون افزایش زیاد سربار کنترل ریسه‌هاست. دو حالت 2 و 8 عملکرد تقریبا برابری داشتند و تمامی حالات چندریسه به مراتب بهتر از تک ریسه عمل کرده‌اند.

حالت دوم بررسی فایل index.html است. نمودارهای زیر به دست آمدند:

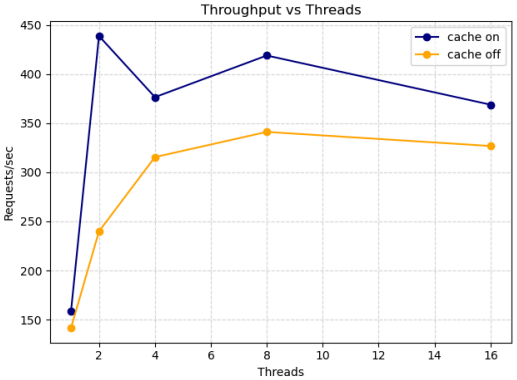
به علت سادگی و سبکی فایل html، این بار نوسان عملکرد بسیار مشخص است. اما در هر دو حالت خاموش یا روشن بودن کش، همچنان عملکرد 4 ریسه از همه حالات بهتر است. اما اگر کش را خاموش کنیم، دیگر بهبود وضعیت چندریسه نسبت به تک ریسه ناچیز می‌شود. البته باید توجه داشت خصوصا در ارتباط با نوسان عجیب در حالت دو ریسه، این اتفاق جالب می‌تواند ناشی از آن

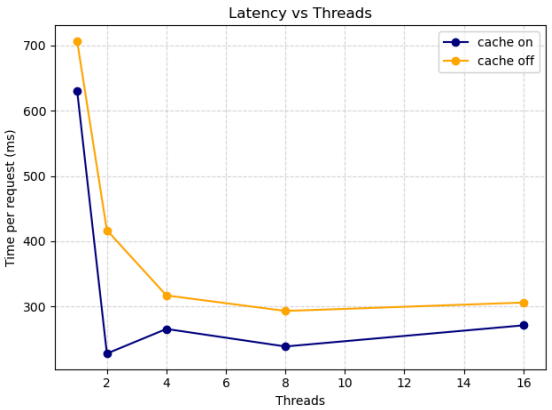
باشد که چون فایل خروجی خیلی کوچک است، کار هر ریسه زود تمام می‌شود. بنابرین خیلی سریع برمیگردند تا عملیات بعدی را بردارند، حال اینجا ممکن است زمان بیشتری نسبت به زمان اجرای خود کارکرد پشت قفل‌ها گیر بکنند و باعث شود عمکرد سیستم بدتر شود.





حالت سوم تست روی فایل باینری بزرگ است. اینجا خیلی بیشتر مطابق انتظار هم عملکرد عالی چندریسه و هم بهبود کش روی تمام حالات را می‌بینیم. در این حالت خاص به نظر می‌رسد برعکس حالت معمول تست اول، ریسه‌های 2 و 8 عملکرد بهتری نسبت به 4 در حالت روشن بودن کش داشته‌اند. تقریبا در این نمودار و تمام نمودارهای بعدی اگر کش را خاموش کنیم عملکرد 8 ریسه بهتر از 4 ریسه است.





در نهایت یک کانفیگ خیلی سبک با سطح همروندی 2 را تست کردیم که نتایج زیر به دست آمد. در این نتایج به خوبی تاثیر عالی کش در افزایش پرفورمنس قابل مشاهده‌است. در دنیای واقعی در بخش‌های زیادی از زمان، دسترسی به یک سرور با سطح همروندی پایینی صورت می‌گیرد(مثلا در شب یا در مواقع کم کاربرد روز) اینجاست که تاثیر کش خیلی خوب خودش را نشان می‌دهد. وقتی همروندی پایین است، تعداد کاربر همزمان کمتر داریم و این یعنی لوکالیتی بیشتر که باعث میشود کش عملکرد بهتری از خودش نشان بدهد.