Linux у вбудованих системах: Yocto, Buildroot, OpenWRT

Ковальов Олександр, ІМ-51мн, ФІОТ

23 лютого 2026 р.

Зміст

1 Вступ **2**

2 Анатомія вбудованої Linux-системи **3**

3 Система збірки Buildroot **4**

4 Проект Yocto **6**

5 Дистрибутив OpenWRT **7**

6 Практичний кейс: розгортання системи аналізу мережевого трафіку **9**

7 Порівняльний аналіз та висновки **10**

1 Вступ

Термін Embedded Linux описує використання операційної системи на базі ядра Linux у вбудованих системах та спеціалізованих пристроях. На відміну від традиційних десктопних або серверних дистрибутивів, вбудований Linux не є універсальним. Це ретельно налаштована, мінімалістична система, яка містить лише ті компоненти, що необхідні для виконання конкретної задачі на специфічному апаратному забезпеченні. До таких пристроїв належать телекомунікаційні маршрутизатори, розумні прилади, промислові контролери та складні системи моніторингу.

Вибір Linux як стандарту де-факто у світі розумних пристроїв та мережевого обладнання зумовлений кількома ключовими факторами. Найголовніше – це відкритий вихідний код та неперевершена апаратна підтримка, що дозволяє запускати систему на різноманітних архітектурах, таких як ARM, MIPS, x86 чи RISC-V. Крім того, наявність зрілого мережевого стеку, сучасних механізмів безпеки та величезної екосистеми готових програмних пакетів робить Linux ідеальним фундаментом для складних рішень. Замість того, щоб розробляти базовий системний функціонал з нуля, інженери можуть зосередитись на створенні унікальної логіки свого продукту, використовуючи надійну та перевірену часом базу.

У цій розгалуженій екосистемі роль розробника прикладного програмного забезпечення для вбудованих систем набуває особливої ваги. Традиційно вбудована розробка асоціювалася переважно з написанням низькорівневого коду для мікроконтролерів (bare-metal). Однак, з розповсюдженням Embedded Linux фокус суттєво зміщується. Розробник (software embedded engineer) отримує у своє розпорядження повноцінну операційну систему, що дозволяє використовувати сучасні та безпечні мови програмування для створення складних застосунків. Основна задача такого спеціаліста полягає не в розробці драйверів чи адаптації ядра під нову плату, а в ефективному використанні ресурсів ОС для вирішення комплексних прикладних задач. Чудовим прикладом такого підходу є розробка систем глибокого аналізу трафіку (DPI), подібних до мого проекту xailyser, про який я розповім пізніше. У таких сценаріях вбудований Linux виступає надійним майданчиком для розгортання високопродуктивного аналітичного програмного забезпечення, що безпосередньо взаємодіє з мережевими підсистемами ядра, залишаючись при цьому апаратно-незалежним.

2 Анатомія вбудованої Linux-системи

Будь-яка вбудована система на базі Linux, незалежно від її складності чи призначення, будується на чотирьох фундаментальних компонентах. Розуміння цих складових є критично важливим для розробника прикладного рівня, оскільки саме вони формують середовище, у якому буде виконуватися фінальний код. Ці чотири елементи тісно взаємодіють між собою, утворюючи цілісну платформу від моменту подачі живлення до запуску специфічних користувацьких застосунків, таких як платформи глибокого аналізу мережевого трафіку.

Першим і найголовнішим інструментом у процесі розробки є набір інструментів крос-компіляції, або крос-тулчейн. Оскільки цільові пристрої часто мають суттєво обмежені ресурси та базуються на архітектурах, відмінних від робочої станції розробника, компілювати код безпосередньо на них непрактично. Тулчейн дозволяє використовувати обчислювальну потужність хост-машини для генерації бінарних файлів, призначених для виконання на цільовому пристрої з архітектурою ARM, MIPS або RISC-V. Для інженера програмного забезпечення це означає можливість комфортно писати високопродуктивний системний код, і компілювати його під специфічне мережеве обладнання, маючи повну впевненість у сумісності архітектур та системних викликів.

Після того як програмне забезпечення скомпільовано, системі потрібен механізм для початкового запуску. Цю відповідальну роль виконує завантажувач операційної системи, серед яких найвідомішим є U-Boot. Його головна задача полягає в ініціалізації базового апаратного забезпечення, такого як контролери оперативної пам'яті та накопичувачів, відразу після увімкнення живлення пристрою. Завантажувач зчитує ядро операційної системи з постійної пам'яті або завантажує його через мережу, розміщує в оперативній пам'яті та передає йому управління. Хоча прикладний програміст рідко взаємодіє із завантажувачем напряму, розуміння його принципів роботи дуже допомагає в процесі оновлення прошивок та відновлення працездатності системи після критичних збоїв.

Ядро Linux є справжнім серцем усієї вбудованої платформи. Воно перебирає на себе управління після завантажувача, ініціалізує всі інші периферійні пристрої та монтує кореневу файлову систему. Для програмного рівня ядро надає стандартизований інтерфейс абстракції апаратного забезпечення. Це означає, що розробнику аналітичного ПЗ не потрібно знати деталі реалізації конкретного мережевого чіпа; йому достатньо використовувати стандартні мережеві сокети або сучасні технології низькорівневого моніторингу, такі як розширені фільтри пакетів Берклі. Саме ядро забезпечує ізоляцію процесів, управління пам'яттю та справедли-

вий розподіл процесорного часу, гарантуючи стабільну роботу складних аналітичних інструментів та постійних фонових сервісів.

Завершальним, але не менш важливим елементом є коренева файлова система, яка зберігає всі необхідні для роботи операційної системи та прикладних програм файли. На відміну від універсальних дистрибутивів, де коренева файлова система може займати десятки гігабайтів, у вбудованому світі вона максимально оптимізована та урізана до найнеобхіднішого мінімуму. Вона містить лише базові системні бібліотеки, утиліти конфігурації мережі, скрипти ініціалізації та, що найважливіше для розробника, самі користувацькі застосунки. Практична розробка під Embedded Linux для software-інженера часто зводиться саме до правильної інтеграції власного програмного забезпечення, такого як системи виявлення загроз або кастомні агенти збору метрик, у цю структуру, забезпечення наявності всіх потрібних динамічних бібліотек та налаштування надійного автоматичного старту під час завантаження пристрою.

3 Система збірки Buildroot

Buildroot представляє собою систему збірку дистрибутиву – еталон простоти та прозорості у світі розробки вбудованих систем на базі Linux. Його фундаментальна філософія побудована навколо звичних та перевірених часом механізмів GNU Make та системи конфігурації Kconfig, тобто тієї самої системи, яка використовується для налаштування самого ядра Linux. Цей архітектурний вибір робить Buildroot надзвичайно зрозумілим для інженерів, які вже мають досвід роботи із системним програмуванням в середовищі Linux. Замість того, щоб впроваджувати складні нові парадигми або важкі багаторівневі рушії, Buildroot покладається на просте дерево файлів Makefile. Коли розробник запускає інтерфейс конфігурації, він бачить єдине текстове меню, яке дозволяє вибрати цільову архітектуру, налаштувати тулчейн, обрати необхідні системні утиліти та вказати, які саме прикладні програми повинні увійти до фінальної кореневої файлової системи. Результатом цього процесу є єдиний конфігураційний файл, який повністю керує процесом збірки від початку і до кінця.

При виборі інструментарію для проекту, Buildroot виступає ідеальним кандидатом для створення вузькоспеціалізованих, обмежених у ресурсах або відносно статичних пристроїв. Його головний патерн проектування обертається навколо створення повноцінного, незмінного образу прошивки. На відміну від настільних чи серверних дистрибутивів, Buildroot за замовчуванням не передбачає наявності пакетного менеджера

на цільовому пристрої. Це означає, що всі необхідні компоненти, починаючи від системних бібліотек і закінчуючи фінальним прикладним програмним забезпеченням, компілюються та пакуються в єдиний образ ще на етапі збірки на робочій станції розробника. Такий підхід робить Buildroot ідеальним інструментом для створення програмного забезпечення для промислових контролерів, спеціалізованих датчиків або компактних систем збору даних. У контексті кібербезпеки та мережевого моніторингу Buildroot дозволяє згенерувати мінімалістичний і захищений образ операційної системи, позбавлений будь-яких зайвих фонових процесів чи утиліт, які могли б споживати цінні ресурси або стати вектором атаки.

Для інженера прикладного програмного забезпечення найбільш інтригуючим аспектом Buildroot є процес інтеграції власного коду в загальну систему. Оскільки середовище повністю базується на стандартах Make, додавання нового застосунку зводиться до створення простого та логічного опису правил його збірки. Структура Buildroot передбачає наявність окремої директорії для кожного системного або користувацького пакета, де розробнику потрібно розмістити лише два ключових файли. Перший файл відповідає за відображення пакета у головному меню конфігурації, що дає змогу легко вмикати або вимикати його під час формування прошивки. Другий файл містить безпосередні інструкції для системи збірки: звідки завантажити вихідний код програми, які системні бібліотеки потрібні для її успішної компіляції та якими саме командами слід запускати процес компіляції. Цей механізм є надзвичайно зручним при роботі із сучасними мовами програмування, дозволяючи легко викликати зовнішні інструменти на кшталт Cargo для проектів на мові Rust або використовувати класичні сценарії збірки для проектів на мові C.

Розглядаючи практичний сценарій розгортання певної системи, Buildroot пропонує максимально прямий і передбачуваний шлях до готового пристрою. Розробнику достатньо описати пакет, який вказує на репозиторій із вихідним кодом. Під час загального запуску процесу збірки системи Buildroot автоматично підготує необхідний крос-тулчейн, стягне код аналізатора, скомпілює його під архітектуру цільового пристрою та коректно розмістить готовий бінарний файл у відповідну директорію майбутньої файлової системи. Крім того, інженер може додати спеціальні конфігураційні файли або стартові скрипти для автоматичного запуску свого застосунку відразу після завантаження операційної системи пристрою. Вся ця логіка описується в межах одного невеликого пакета і стає невід'ємною частиною надійно відтворюваного процесу збірки, що гарантує абсолютну ідентичність поведінки прошивки при кожному новому релізі програмного забезпечення.

4 Проект Yocto

Проект Yocto кардинально відрізняється від традиційного розуміння операційної системи. Це не готовий дистрибутив у класичному розумінні, а надзвичайно потужний та гнучкий фреймворк для створення власних, глибоко кастомізованих дистрибутивів Linux. Сучасна індустрія, від автомобілебудування до телекомунікаційних гігантів, недарма обирає Yocto як беззаперечний стандарт де-факто для розробки складних, масштабованих enterprise-рішень. Основна причина такого вибору полягає в тому, що цей інструментарій забезпечує безпрецедентний рівень гнучкості, відтворюваності та абсолютного контролю над кожним компонентом системи, починаючи від налаштувань ядра і закінчуючи розгортанням специфічного прикладного програмного забезпечення на тисячах однотипних пристроїв.

Серцем екосистеми Yocto та її головним рушієм є планувальник завдань BitBake. Це складна система, яка концептуально нагадує утиліту Make, але створена спеціально для управління гігантськими та заплутаними графами залежностей під час крос-компіляції цілої операційної системи. BitBake аналізує тисячі інструкцій, автоматично визначає оптимальний порядок завантаження вихідного коду, накладання специфічних патчів, конфігурації середовища, компіляції та пакування готових бінарних файлів у формати на кшталт RPM або IPK. Для інженера-розробника прикладного рівня, який працює з продуктивними мовами на кшталт C або Rust, BitBake виступає надійним абстрактним шаром. Він бере на себе всю складну рутину з налаштування правильного середовища крос-компіляції, гарантуючи, що системні виклики, бібліотеки та оптимізації цільової апаратної архітектури будуть застосовані абсолютно коректно.

Базовою одиницею конфігурації та логіки в системі Yocto є рецепт. Рецепти являють собою спеціальні текстові файли, які містять вичерпну та формалізовану інформацію про те, як саме потрібно обробити конкретний програмний компонент. Вони вказують рушію BitBake мережеві координати вихідного коду, перелік необхідних залежностей для успішної збірки, специфічні параметри компіляції та точні шляхи для розміщення результату у фінальній кореневій файловій системі. Процес написання рецепту для власного застосунку, наприклад, для кастомного компонента платформи глибокого аналізу пакетів, вимагає розуміння життєвого циклу збірки вашого коду. На щастя, Yocto має чудову вбудовану підтримку інтеграції сучасних систем збірки, таких як Cargo для проектів мовою Rust або CMake для архітектур мовою C, що суттєво спрощує перенесення високопродуктивного аналітичного коду в суворе

середовище вбудованого Linux.

Найбільшою архітектурною перевагою Yocto, яка робить його незамінним в enterprise-середовищі, є концепція багаторівневих шарів. Шари дозволяють логічно та фізично розділити метадані всього проекту. Наприклад, один базовий шар може містити специфічні рецепти для підтримки апаратного забезпечення від виробника процесора, інший шар може визначати корпоративні політики безпеки та мережеву конфігурацію ядра, а третій шар, розроблений вашою командою, буде містити рецепти виключно для вашого прикладного програмного забезпечення. Ця модульна архітектура дозволяє неймовірно легко портувати розроблені рішення на абсолютно нове обладнання. Як приклад, якщо ви створили власний шар з, наприклад, рецептами для системи децентралізованої кіберрозвідки або аналізатора трафіку, тощо, ви можете просто підключити цей шар до нової конфігурації Yocto для іншого маршрутизатора чи шлюзу, якщо ми говоримо про мережеве забезпечення. BitBake автоматично перекомпілює ваш прикладний код під нову архітектуру, зберігаючи всі налаштування програмної логіки повністю ізольовано від низькорівневої апаратної частини.

Масштабованість рішень на базі Yocto яскраво проявляється і в управлінні життєвим циклом програмного забезпечення. Інструментарій дозволяє генерувати точні маніфести ліцензій, створювати інфраструктуру для безпечних оновлень системи по повітрю та безшовно інтегрувати складні підсистеми моніторингу. Для дослідницьких та комерційних проектів, що фокусуються на аналізі мереж або використанні інноваційних технологій на кшталт eBPF, Yocto дає змогу надзвичайно гнучко конфігурувати ядро, активуючи лише необхідні модулі та підсистеми. Це створює ідеальне, мінімалістичне та захищене середовище, де прикладне програмне забезпечення працює на максимально оптимізованій платформі, що є критичною вимогою для обробки величезних масивів мережевих пакетів у сучасних телекомунікаційних та безпекових комплексах.

5 Дистрибутив OpenWRT

Переходячи до OpenWRT, ми стикаємося з абсолютно іншою парадигмою в екосистемі вбудованого Linux. Якщо Buildroot фокусується на статичних монолітних образах, а Yocto – на масштабованому створенні власних дистрибутивів, то OpenWRT вже є повноцінним, високоспеціалізованим дистрибутивом, створеним спеціально для мережевого обладнання. Його архітектура від самого початку проектувалася з урахуванням потреб маршрутизаторів, шлюзів та точок доступу, де критично важли-

во ефективно обробляти мережевий трафік в умовах жорстко обмежених ресурсів процесора та оперативної пам'яті. Для розробника програмного забезпечення це означає, що замість боротьби з базовою інфраструктурою він одразу отримує ідеально налаштоване середовище з оптимізованим мережевим стеком, де всі системні компоненти працюють на забезпечення максимальної пропускної здатності.

Однією з найважливіших особливостей OpenWRT, яка кардинально відрізняє його від багатьох інших вбудованих рішень, є наявність повноцінної системи управління пакетами opkg. Цей інструмент працює за принципами, дуже схожими на apt у Debian або dnf у Fedora, але оптимізований для компактного зберігання даних. Завдяки opkg прошивка пристрою перестає бути монолітною структурою, яку потрібно повністю перепрошивати при кожній найменшій зміні. Розробник або навіть кінцевий користувач може динамічно встановлювати, оновлювати або видаляти програмне забезпечення безпосередньо на працюючому пристрої через мережу. Для створення і тестування складних систем, таких як платформи аналізу трафіку, це є беззаперечною перевагою, оскільки дозволяє миттєво доставляти нові версії бінарних файлів на роутер без переривання його основної роботи.

Специфіка мережевого обладнання вимагає зручного і стандартизованого підходу до конфігурації численних інтерфейсів, брандмауерів та правил маршрутизації. В OpenWRT це реалізовано через Уніфікований Інтерфейс Конфігурації (UCI). Замість того, щоб редагувати десятки різних конфігураційних файлів у форматах, специфічних для кожної окремої утиліти, розробник взаємодіє з єдиною стандартизованою системою текстових файлів або використовує зручні консольні команди. UCI автоматично трансформує ці загальні налаштування у специфічні параметри для системних демонів (служб) під час їхнього запуску. Це критично важливо для інтеграції власних мережевих застосунків, оскільки дозволяє легко програмно маніпулювати правилами перенаправлення пакетів або налаштуваннями інтерфейсів, щоб направити потік даних безпосередньо у ваш застосунок.

Коли мова йде про розробку власного прикладного програмного забезпечення для OpenWRT, розробник неминуче стикається з проблемою вибору архітектури керуючого інтерфейсу. Традиційним шляхом є створення модулів для стандартного веб-інтерфейсу маршрутизатора LuCI, який написаний мовою Lua. Хоча цей підхід добре інтегрується в існуючу екосистему роутера, він накладає значні обмеження на продуктивність і є поганим вибором для ресурсоємних завдань або складної візуалізації даних у реальному часі. Іншою сучасною альтернативою є використання технології WebAssembly (WASM) у комбінації з локальним веб-сервером

на пристрої, що дозволяє перенести частину логіки в браузер клієнта. Проте, найкращим архітектурним рішенням для інтенсивних обчислень є повне розділення логіки обробки та візуалізації.

Саме цей підхід, який базується на розгортанні виділеного високопродуктивного сервера на пристрої та віддаленому підключенні до нього, був успішно застосований у проекті xailyser, який я написав для моэъ дипломної бакалаврської роботи. У цій архітектурі вбудований пристрій під управлінням OpenWRT виконує лише те, що він робить найкраще: перехоплює, фільтрує та аналізує мережевий трафік за допомогою низькорівневого коду. Паралельно на ньому працює легкий сервер, який надає API для доступу до результатів аналізу. Керування ж системою та складна візуалізація метрик здійснюється через віддалений клієнт, запущений на потужній робочій станції користувача. Такий дизайн не перевантажує слабкий процесор маршрутизатора задачами рендерингу або підтримки важких веб-сесій, гарантуючи, що всі системні ресурси спрямовані виключно на критично важливу задачу мережевого моніторингу без втрати пакетів.

6 Практичний кейс: розгортання системи аналізу мережевого трафіку

Я б хотів би розповісти про те, як в ідеалі моя система глибокого аналізу мережевого пакетного трафіку (DPI) повинна була деплоїтися на маршрутизатор, бо я використовував крос-компіляцію бінарника та заливання його по ssh. Головна складність таких проектів полягає в тому, що програмне забезпечення повинно працювати безпосередньо «на залізі» маршрутизатора, маючи мінімальні затримки при доступі до мережевих інтерфейсів, але при цьому воно має бути написане з використанням сучасних, безпечних інструментів, таких як мова програмування Rust. Оскільки розробка ведеться на потужній робочій станції, першим критичним кроком стає налаштування середовища крос-компіляції. Замість того, щоб намагатися зібрати весь дистрибутив з нуля, інженер завантажує попередньо скомпільований OpenWRT SDK для конкретної архітектури цільового процесора. Це середовище містить готовий тулчейн та копію кореневої файлової системи пристрою з усіма необхідними заголовковими файлами.

Процес компіляції складного аналізатора вимагає ретельного управління системними залежностями. Проекти рівня DPI зазвичай покладаються на специфічні бібліотеки перехоплення трафіку, такі як libpcap,

або потребують доступу до структур ядра для роботи з eBPF. OpenWRT SDK дозволяє елегантно вирішити цю проблему шляхом створення власного конфігураційного пакету, який вказує системі збірки стягнути код аналізатора та автоматично злінкувати його з правильними версіями динамічних бібліотек для цільової архітектури. Результатом цього автоматизованого процесу стає стандартний інсталяційний файл з розширенням IPK. Цей підхід гарантує, що скомпільований бінарний файл буде абсолютно сумісним із системним оточенням маршрутизатора і не вимагатиме жодних ручних маніпуляцій із файлами бібліотек на самому пристрої.

Фінальним етапом інтеграції є перетворення нашого застосунку на повноцінний системний сервіс. У середовищі OpenWRT для управління фоновими процесами використовується підсистема procd. Розробник додає до свого інсталяційного пакету спеціальний скрипт ініціалізації, який вказує системі автоматично запускати сервер xailyser під час завантаження маршрутизатора, відстежувати його стан та перезапускати у разі непередбачуваних збоїв. Найбільшою перевагою такого підходу є повна автоматизація роботи. Після встановлення пакета через менеджер opkg, система аналізу трафіку починає працювати у фоновому режимі як надійний демон. Вона самостійно збирає метрики та відкриває мережевий порт для API. Це архітектурне рішення повністю усуває необхідність працювати в терміналі цільового пристрою. Взаємодія з системою відбувається виключно через зручний віддалений клієнт, який підключається до сервера аналізатора, дозволяючи зосередитися на візуалізації та аналізі даних без ручного адміністрування консолі.

7 Порівняльний аналіз та висновки

Підсумовуючи розгляд трьох провідних підходів до створення вбудованих Linux-систем, важливо зазначити, що між ними немає абсолютної конкуренції. Вибір між Buildroot, проектом Yocto та дистрибутивом OpenWRT завжди є компромісом між простотою впровадження, необхідним рівнем гнучкості та кінцевим призначенням самого пристрою. Кожен з цих інструментів займає власну нішу і вирішує специфічний клас інженерних проблем, формуючи унікальне середовище для розгортання прикладного коду.

Система Buildroot є абсолютним лідером там, де потрібна максимальна простота та швидкість старту розробки. Її архітектура на базі статичних файлів конфігурації ідеально підходить для невеликих команд та монолітних проектів. Якщо метою є створення вузькоспеціалізованого

датчика, простого промислового контролера або компактного пристрою, де набір програмного забезпечення ніколи не буде змінюватися після випуску з фабрики, Buildroot дозволить згенерувати мінімалістичну та безпечну прошивку за рекордно короткий час. Проте цей інструмент стає незручним, коли проект розростається до масштабів великої продуктової лінійки з різними апаратними платформами.

Проект Yocto знаходиться на протилежному кінці спектра складності. Це важковаговик корпоративного рівня, який вимагає значних інвестицій часу у вивчення своїх концепцій, таких як рецепти та багаторівневі шари. Однак ця крута крива навчання повністю виправдовує себе в довгостроковій перспективі для масштабних проектів. Yocto є незамінним, коли компанії потрібно підтримувати десятки різних пристроїв на абсолютно різних процесорних архітектурах, маючи при цьому єдину кодову базу прикладного програмного забезпечення. Для інженера це гарантія того, що його системний код, написаний на сучасних мовах штибу C чи Rust, буде бездоганно портований на будь-яке нове обладнання шляхом простого підключення відповідного апаратного шару.

Дистрибутив OpenWRT займає унікальну позицію, будучи повністю сфокусованим на мережевому обладнанні. Він звільняє розробника від необхідності збирати базову операційну систему з нуля, надаючи готовий оптимізований фундамент із динамічним менеджером пакетів. Це найкращий вибір для маршрутизаторів, шлюзів безпеки та платформ глибокого аналізу трафіку. Для спеціаліста, що займається розробкою програмного забезпечення на рівні застосунків, OpenWRT забезпечує найкоротший шлях до запуску коду на реальному мережевому залізі.

У підсумку, роль сучасного розробника вбудованого програмного забезпечення полягає не лише у написанні ефективних алгоритмів, але й у правильному виборі платформи для їхнього виконання. Використання таких інструментів дозволяє абстрагуватися від низькорівневої рутини, інтегрувати передові мови програмування та створювати надійні, автоматизовані сервіси. Завдяки правильній архітектурі, як це було продемонстровано на прикладі системи xailyser, можна побудувати складні аналітичні комплекси, управління якими відбувається виключно через сучасні віддалені клієнти, залишаючи роботу в терміналі пристрою в минулому і фокусуючись виключно на розвитку логіки продукту.