Реферат

На тему: ***«Побудова надійних операційних систем, що допускають наявність ненадійних драйверів пристроїв»***

**Введення**

Найбільш гострою проблемою багатьох користувачів є ненадійність комп'ютерів.

Дослідники у галузі комп'ютерної науки звикли до регулярних збоїв комп'ютерів і до необхідності через кожні кілька місяців встановлювати патчі програмного забезпечення. Проте переважна більшість користувачів вважає це відсутність надійності неприйнятним. Їхня внутрішня модель роботи електронного пристрою ґрунтується на досвіді використання телевізорів і відеомагнітофонів: ви купуєте пристрій, підключаєте його до мережі, і воно бездоганно працює протягом 10 років. Ніяких відмов, ніяких регулярних оновлень програмного забезпечення, ніяких газетних історій про виявлення новітніх представників нескінченної низки вірусів. Щоб зробити комп'ютерні системи більш схожими на телевізори, ми ставимо за мету свого дослідження вдосконалення надійності комп'ютерних систем, і починаємо з операційних систем.

**1. Чому у систем трапляються відмови?**

Основна причина аварійних відмов операційних систем криється у двох принципових дефекти розробки, властивих всім цим системам: наявність занадто великого числа привілеїв і відсутність адекватної ізоляції збоїв. Практично всі операційні системи складаються з численних модулів, скомпонованих в одному адресному просторі і утворюють єдину бінарну програму, яка виконується в режимі ядра. Помилка в будь-якому модулі може легко призвести до руйнування структур даних в будь-якому іншому, не пов'язаним з ним модулі і до миттєвого виходу системи з ладу. Причиною, за якою всі модулі компонуються в єдиний адресний простір без підтримки будь-якої захисту між модулями, є Фаустова угода розробників: покращена продуктивність за ціну більшого числа відмов системи. Нижче ми оцінимо вартість цього компромісу.

Тісно пов'язаний питання відноситься до першопричину аварійних відмов. Адже якби кожен модуль був бездоганним, то не виникала б потреба в ізоляції збоїв між модулями, оскільки не було б самих збоїв. Ми стверджуємо, що більша частина збоїв виникає через помилки програмування, внаслідок надмірної складності і використання чужого коду. Дослідження показують, що в програмному забезпеченні в середньому міститься від однієї до шістнадцяти помилок на тисячу рядків коду [27, 22, 2], і що верхня межа цього діапазону явно занижена, оскільки враховувалися тільки ті помилки, які, врешті-решт, вдавалося виявити. Очевидним висновком є те, що в більшому обсязі коду міститься більша кількість помилок. У міру розвитку програмного забезпечення в кожній його новій версії з'являється все більше можливостей (і, відповідно, більший об'єм коду), і часто нова версія є менш надійною, ніж попередня. У [22] показано, що число помилок на тисячу рядків коду прагне до стабілізації у міру зростання числа випущених версій, але асимптотично цей показник відрізняється від нуля.

Наявність деяких з цих помилок дозволяє зловмисникам застосовувати віруси і черв'яки для зараження і пошкодження системи. Так що деякі нібито наявні проблеми «безпеки» в принципі не мають нічого спільного з порушеннями заходів безпеки (наприклад, дефектними криптографічними алгоритмами або нестійкими протоколами авторизації), а викликаються лише помилками в коді програм (наприклад, переповнення буферів дозволяють виконувати впроваджений код). Коли в цій статті ми говоримо про «надійності», ми маємо на увазі й те, що часто називають «безпекою», – неавторизований доступ внаслідок помилки в коді програми.

Друга проблема полягає в привнесення в операційну систему чужого коду. Найбільш досвідчені користувачі ніколи б не дозволили сторонньої організації вставити незнайомий код в ядро операційної системи, хоча, коли вони купують нове периферійне пристрій і інсталюють відповідний драйвер, вони саме це й роблять. Драйвери пристроїв звичайно пишуться програмістами, що працюють на виробників периферійних пристроїв, і контроль якості їх продукції звичайно нижче, ніж у постачальників операційних систем. У тих випадках, коли драйвер відноситься до open-source, його часто пише благонамірений, але не обов'язково досвідчений доброволець, і контроль якості забезпечується на ще більш низькому рівні. Наприклад, в Linux частота появи помилок в драйверах пристроїв від трьох до семи разів вище, ніж в інших частинах ядра [7]. Навіть компанія Microsoft, у якої є стимули та ресурси для застосування більш щільного контролю якості, не може добитися набагато кращих результатів: 85% всіх аварійних відмов Windows XP обумовлюється наявністю помилок у коді драйверів.

Останнім часом з'явилися публікації про родинні роботах, присвячених ізоляції драйверів пристроїв з використанням апаратури MMU [26] і віртуальних машин [19]. Ці методи концентруються на вирішенні проблем у успадкованих операційних системах; ми обговоримо їх у розд. 6. На відміну від цього, при застосуванні нашого підходу надійність досягається шляхом розробки нової полегшеної операційної системи.

**2. Рішення: правильна ізоляція збоїв**

Протягом десятиліть як перевірений методу оперування кодом, що не заслуговує довіри, використовувалося розміщення його в окремому процесі та виконання в режимі користувача. Одним з ключових спостережень, отриманих у дослідженні, якому присвячена ця стаття, є те, що потужним засобом підвищення надійності операційної системи є виконання кожного драйвера у вигляді окремого процесу в режимі користувача з мінімальними необхідними привілеями. Таким чином, код, потенційно містить помилки, ізолю, і помилка, скажімо, в драйвері принтера може призвести до припинення друку, але не до запису перекручених даних у будь-які важливі структури даних ядра і виходу системи з ладу.

У цій статті ми проводимо ретельне відмінність між крахом операційної системи, після якого потрібне перезавантаження комп'ютера, і збоєм або відмовою сервера або драйвера, після якого в нашій системі перезавантаження не потрібно. У багатьох випадках дефектний драйвер, що виконується в режимі користувача, може бути вилучений і замінений без потреби в перезапуску інших частин операційної системи, які виконуються в режимі користувача.

Ми не розраховуємо на те, що незабаром з'явиться код, вільний від помилок, а якщо і з'явиться, то, звичайно, не в операційних системах, які зазвичай пишуться на C або C + +. На жаль, у програмах, написаних на цих мовах, інтенсивно використовуються покажчики, рясний джерело помилок. Тому наш підхід заснований на ідеях модульності та ізоляції збоїв. Шляхом розбиття системи на велику кількість ізольованих модулів, кожен з яких виконується в окремому процесі в режимі користувача, нам вдалося скоротити частину системи, виконувану в режимі ядра, до абсолютного мінімуму і запобігти поширенню збоїв, що виникають в інших модулях. Зменшення розмірів ядра значно скорочує число помилок, які воно, ймовірно, має містити. Малий розмір також дозволяє знизити рівень складності ядра і полегшити його розуміння, що також сприяє надійності. Тому ми пішли максими Сент-Екзюпері і зробили ядро настільки невеликим, наскільки це дозволяють людські можливості: менше 3800 рядків коду.

Одне із зауважень, постійно виникає з приводу таких розробок мінімального ядра, стосується уповільнення роботи системи через додаткові перемикань контексту і копіювання даних, яке потрібно для забезпечення комунікацій різних моделей, які виконуються в користувацькому адресному просторі. Це побоювання, в основному, існує з історичних причин, і ми стверджуємо, що ці причини, більшою частиною, наразі відсутні. По-перше, результати нових досліджень показують, що розробка мінімального ядра не обов'язково завдає шкоди ефективності [3, 23, 15]. Зменшення розмірів ядра при наявності розумних протоколів взаємодії серверів допомагає обмежити масштабність проблеми ефективності. По-друге, значне зростання потужності комп'ютерів в останнє десятиліття істотно послаблює проблему гарантованої продуктивності, що виникає при модульної розробці. По-третє, ми вважаємо, що настає час, коли велика частина користувачів з задоволенням пожертвує деякої ефективністю задля поліпшеної надійності.

Детальний обговорення ефективності нашої системи ми представляємо в розд. 5. Однак тут ми коротко згадаємо три попередніх показника ефективності на підтримку нашого доводу про те, що системи з мінімальним ядром не обов'язково повинні бути повільними. По-перше, виміряний час виконання найпростішого системного виклику getpid складає 1.01 мсек на процесорі Athlon з частотою 2.2 Ггц. Це означає, що програма, яка виробляє 10000 системних викликів в секунду, витрачає на перемикання контексту всього 1% часу ЦП, а 10000 системних викликів в секунду виробляють лише деякі програми. По-друге, наша система здатна протягом 4 секунд повністю провести свою компоновку, включаючи ядро і всі частини, що виконуються в режимі користувача (при цьому компілюються 123 файлу і відбувається 11 редагувань зв'язків). По-третє, час початкового завантаження системи з моменту виходу з монітора багатоваріантної завантаження до видачі запрошення до входу в систему становить менше 5 секунд. Після цього операційна система, повністю сумісна з POSIX, готова до використання.

**3. Вклад цієї статті**

Дослідження, результати якого описуються в цій статті, було направлено на вироблення відповіді на таке запитання: як уникнути ситуацій, в яких серйозна помилка в драйвері пристрою (наприклад, використання невірного покажчика або наявність нескінченного циклу) призводить до аварійного відмови або зависання всієї операційної системи?

Наш підхід полягав у розробці надійної мультисерверного операційної системи поверх крихітного ядра, що не містить будь-якого зовнішнього, ненадійного коду. Для забезпечення належної ізоляції збоїв кожен сервер і драйвер виконується в режимі користувача в рамках окремого процесу. Крім того, ми додали механізми для відновлення після виникнення поширених збоїв. Ми детально описуємо засоби підтримки надійності і пояснюємо, чому вони відсутні у традиційних монолітних операційних системах. Ми також обговорюємо отримані показники ефективності системи і показуємо, що кошти підтримки надійності сповільнюють систему на 5–10%, але роблять її стійкою до наявності невірних покажчиків, нескінченних циклів і інших помилок, які призвели б до аварійного відмови або зависання традиційних операційних систем.

Хоча ні один з окремих аспектів нашого підходу (ядра невеликого розміру, драйвери пристроїв, які виконуються в режимі користувача, або мультисерверного системи) не є новим, ніхто раніше не збирав до купи всі ці частини для побудови невеликий, гнучкою, модульної UNIX-подібної системи, що є набагато більш відмовостійкої, ніж звичайні системи сімейства UNIX, і втрачає тільки 5–10% ефективності порівняно з нашою базовою системою, яка містить драйвери в ядрі.

Крім того, наш підхід у корені відрізняється від інших аналогічних робіт, оскільки ми не фокусуємося на масових операційних системах. Замість цього ми отримуємо надійність на основі нової, полегшеною архітектури. Замість того щоб додавати допоміжний код, який підвищує надійність ненадійних систем, ми розщеплює операційну систему на невеликі компоненти й досягаємо надійності за рахунок модульності системи. Хоча наші методи незастосовні до успадкованим операційним системам, ми сподіваємося, що вони допоможуть зробити більш надійними майбутні операційні системи.

Ми починаємо статтю з порівняння нашої розробки зі структурами інших операційних систем (розд. 2) і далі переходимо до спільному обговоренню засобів підтримки надійності нашої системи (розд. 3). Потім ми аналізуємо надійність (розд. 4) і ефективність (розд. 5) системи на основі реальних вимірів. У кінці статті ми аналізуємо деякі суміжні роботи (розд. 6) і представляємо свої висновки (розд. 7).

**4. Розробка операційної системи**

Цей проект присвячений побудові більш надійної операційної системи. Перш ніж докладно описувати свою розробку, ми коротко обговоримо, яким чином вибір структури операційної системи може безпосередньо впливати на її надійність. У своїх цілях ми будемо проводити розходження між двома структурами операційних систем: монолітними системами і системами з мінімальним ядром. Існують і інші типи операційних систем, такі як екзоядра [10] і віртуальні машини [24]. Вони не мають безпосереднього відношення до даної статті, але ми повернемося до них у розд. 6.

Проблеми монолітних систем

Як показано на рис. 1, у стандартній монолітної системі ядро містить всі операційну систему, скомпоновану в єдиному адресному просторі і виконувану в режимі ядра. Ядро може бути структуровано на компоненти, або модулі, показані на малюнку у вигляді прямокутників з пунктирними сторонами, але між компонентами відсутні захисні кордону. На відміну від цього, прямокутники із суцільними сторонами відповідають окремим процесам, що виконуються в режимі користувача; кожен з цих процесів виконується в окремому адресному просторі, що захищається апаратурою MMU (Memory Management Unit, пристрій управління пам'яттю).

З монолітними операційними системами пов'язана низка проблем, властивих їх архітектурі. Хоча деякі з цих проблем вже згадувалися у введенні, ми наведемо тут їх зведення:

1. Відсутня належна ізоляція збоїв.
2. Весь код виконується на найвищому рівні привілейованості.
3. Величезний розмір коду припускає наявність численних помилок.
4. У ядрі присутній ненадійний сторонній код.
5. Складність систем утрудняє їх супровід.

Цей список властивостей ставить під сумнів надійність монолітних систем. Важливо розуміти, що ці властивості виникають не унаслідок поганої реалізації, а являють собою фундаментальні проблеми, пов'язані з архітектурою операційної системи.

Передбачається коректність ядра, у той час, як тільки лише його розмір означає, що воно має містити численні помилки [27, 22, 2]. Більш того, для всіх операційних систем, в яких код виконується на найвищому рівні привілейованості, і не забезпечується належне стримування поширення збоїв, будь-яка помилка може стати фатальною. Наприклад, неправильно працюючий драйвер пристрою, наданий стороннім розробником, може легко зруйнувати ключові структури даних і вивести з ладу всю систему. Реальність такої загрози випливає з того спостереження, що аварійні відмови більшості операційних систем трапляються з вини драйверів пристроїв [7, 25]. Додатковою проблемою є те, що величезний розмір монолітних ядер робить їх дуже складними і важко розуміти. Без загального розуміння ядра навіть досвідчений програміст може легко внести помилки за рахунок недостатньої поінформованості про побічні ефекти своїх дій.

Системи з мінімальним ядром

На іншому полюсі знаходиться мінімальне ядро, що містить лише чистий механізм і ніякої політики. Мінімальна ядро включає обробники переривань, механізм для запуску та зупинки процесів (шляхом завантаження регістрів MMU і ЦП), планувальник і механізм підтримки міжпроцесної комунікацій; в ідеальному випадку більше в ядро не входить нічого. Підтримка функціональних можливостей стандартної операційної системи, представлених у монолітному ядрі, переміщається в користувальницьке адресний простір, і відповідний код більше не виконується на найбільш привілейованому рівні.

Поверх мінімального ядра можливі різні організації операційної системи. Одним з варіантів є виконання всієї операційної системи в одному сервері в режимі користувача, але в такій архітектурі існують ті ж проблеми, що і в монолітній системі, і помилки, як і раніше можуть призвести до аварійного відмови всієї операційної системи, що виконується в режимі користувача. У розд. 6 ми обговоримо деякі роботи в цій області.

Кращим рішенням є виконання кожного ненадійного модуля в режимі користувача в окремому процесі, ізольованому від інших процесів. Ми до крайності захопилися цією ідеєю і повністю роздрібнили свою систему, як показано на рис. 2. Усі функціональні компоненти операційної системи, такі як драйвери пристроїв, файлова система, сервер мережі та високорівневе управління пам'яттю, виконуються як окремі процеси в режимі користувача у власному адресному просторі. Цю модель можна визначити, як мультисерверного операційну систему.

З логічної точки зору наші користувальницькі процеси можна розбити на три рівні, хоча з точки зору ядра всі вони є всього лише процесами. Найнижчий рівень процесів, які виконуються в режимі користувача, займають драйвери пристроїв, кожен з яких керує деякими пристроєм. Ми реалізували драйвери для інтерфейсу IDE, гнучких і жорстких дисків, клавіатури, дисплеїв, аудіо-пристроїв, принтерів і різних карт Ethernet. Вище рівня драйверів знаходяться серверні процеси. У їх число входять файловий сервер, сервер процесів, мережевий сервер, інформаційний сервер, сервер реінкарнації та інші. Над рівнем серверів виконуються звичайні користувальницькі процеси, включаючи різні інтерпретатори shell, компілятори, утиліти та прикладні програми. Не рахуючи невеликого числа виключень, сервери і драйвери є нормальними для користувача процесами.

Щоб уникнути будь-якої неясності ще раз зауважимо, що кожний сервер або драйвер виконується у вигляді окремого користувача процесу з власним адресним простором, повністю відокремленим від адресного простору ядра і інших серверів, драйверів і процесів користувачів. У нашій архітектурі процеси не поділяють будь-яке адресний простір і можуть спілкуватися один з одним лише з використанням механізму IPC, забезпечуваного ядром. Цей аспект є критичним для надійності, оскільки він запобігає поширенню збоїв одного сервера або драйвера на інші сервери або драйвери подібно до того, як помилка при компіляції програми, що виникає в одному процесі, не впливає на те, що робить браузер в іншому процесі.

Під час роботи в режимі користувача можливості процесів операційної системи обмежені. Тому для підтримки виконання необхідних від них завдань серверами і драйверами ядро експортує ряд системних викликів, які можуть вироблятися авторизованими процесами. Наприклад, драйвери пристроїв більше не мають привілеїв на безпосереднє виконання вводу-виводу, але можуть вимагати від ядра виконання відповідних дій від свого імені. Крім того, сервери та драйвери можуть запитувати сервіси один в одного. Всі такі IPC проводяться шляхом обміну невеликими повідомленнями фіксованого розміру. Цей обмін повідомленнями реалізується шляхом звернень до ядра, яке до виконання запитуваної дії перевіряє, авторизований чи відповідним чином викликає процес.

Розглянемо типовий виклик ядра. Компоненту операційної системи, що виконується в режимі користувача в деякому процесі, може знадобитися скопіювати дані в інше адресний простір чи з нього, але йому неможливо довірити можливість доступу до фізичної пам'яті. Натомість цього забезпечуються виклики ядра для копіювання з допустимих віртуальних адрес або в ці адреси сегмента даних цільового процесу. Цей виклик надає набагато більш слабкі можливості, ніж запис в будь-яке слово фізичної пам'яті, але все-таки ці можливості досить потужні, і тому можливість такого виклику надається тільки процесам операційної системи, яким потрібно копіювання блоків даних з одного адресного простору в інше. Для звичайних користувальницьких процесів подібні виклики заборонені.

Після приведення цього опису структури операційної системи ми можемо тепер пояснити, яким чином користувальницькі процеси отримують сервіси операційної системи, визначені в стандарті POSIX. Користувальницький процес, який бажає виконати, наприклад, виклик READ, формує повідомлення, що містить номер системного виклику і (покажчики на) параметри, і звертається до ядра із запитом посилки цього невеликого запитної повідомлення файлового сервера, що є іншим призначеним для користувача процесом. Ядро забезпечує блокування викликає процесу до тих пір, поки його запит не буде опрацьовано файловим сервером. За замовчуванням усі комунікації між процесами забороняються з міркувань безпеки, але цей запит досягає мети, оскільки комунікації з файловим сервером явно вирішуються звичайним користувальницьким процесам.

Якщо запитувані містяться в буферному кеші файлового сервера, то він виробляє виклик ядра із запитом копіювання цих даних в буфер користувача. Якщо у файлового сервера відсутні необхідні дані, то він посилає повідомлення дисковому драйверу із запитом потрібного блоку. Тоді дисковий драйвер видає команду диска на читання цього блоку прямо за адресою всередині буферного кешу файлового сервера. Коли передача даних з диска завершується, дисковий драйвер посилає файлового серверу повідомлення у відповідь, що містить стан запиту (успіх або причина невдачі). Після цього файловий сервер робить виклик ядра із запитом копіювання блоку в користувальницьке адресний простір.

Ця схема проста і елегантна, вона дозволяє відокремити сервери і драйвери від ядра і дозволяє замінювати їх простим чином, що сприяє модульності системи. Хоча тут потрібно до чотирьох повідомлень, вони передаються дуже швидко (в межах 500 наносекунд на повідомлення в залежності від ЦП). Якщо і відправник, і одержувач готові до комунікації, те ядро копіює повідомлення прямо з буфера відправник у буфер одержувача без його переміщення в адресний простір ядра. Крім того, число копіювань даних є точно таким же, як в монолітній системі: диск поміщає дані прямо в буферний кеш файлового сервера, та є одне копіювання з цього кеша в адресний простір користувацького процесу.

Принципи розробки

Перш ніж перейти до докладного розгляду властивостей надійності нашої системи, коротко обговоримо принципи розробки, якими ми керувалися у прагненні до надійності:

1. Простота.
2. Модульність.
3. Найменша авторизація.
4. Відмовостійкість.

По-перше, ми зберігаємо свою систему настільки простий, наскільки це можливо, так що її легко зрозуміти, і можна з більшою вірогідністю підтримувати її в коректному стані. Це відноситься як до високорівневих проектування, так і до реалізації. Наша розробка дозволяє структурно уникнути відомих проблем, таких як вичерпання ресурсів. При потребі ми явно обмінюємо ресурси та ефективність на надійність. Наприклад, в ядрі статично оголошуються всі структури даних замість того, щоб динамічно виділяти пам'ять при необхідності. Хоча ми можемо недоіспользовать деяку пам'ять, цей підхід є дуже простим і ніколи не призводить до помилок. Іншим прикладом є те, що ми навмисне не реалізували нитки. Може бути, ми заплатили за це деякою втратою ефективності (а може бути, і ні), але зате не повинні турбуватися про потенційних «станах гонок» (race condition) і синхронізації, що істотно полегшує життя програмістам.

По-друге, ми розділили свою систему на набір невеликих незалежних модулів. Використання властивостей модульності, таких як обмеження розповсюдження збоїв, є ключовим елементом розробки нашої системи. Шляхом повного поділу операційної системи на модулі ми можемо встановити «брандмаери», крізь які не можуть розповсюджуватися помилки, що призводить до більш надійної системи. Для запобігання непрямого впливу збоїв в одному модулі на який-небудь інший модуль ми структурним чином зменшуємо їх взаємозалежність, наскільки це можливо. У тих випадках, коли це неможливо через природи модулів, ми застосовуємо додаткові засоби підтримки безпеки. Наприклад, файлова система залежить від драйверів пристроїв, але вона розробляється таким чином, щоб бути готовою до обробки збоїв драйвера.

По-третє, ми забезпечуємо дотримання принципу найменшої авторизації. Хоча ізоляція збоїв допомагає стримувати їх поширення, збій у повноважному модулі все ще може викликати значний збиток. Тому ми знижуємо рівень привілеїв всіх користувальницьких процесів до гранично припустимого мінімуму. У ядрі підтримуються бітові масиви і списки, які визначають можливості процесів. Зокрема, є шкала допустимих викликів ядра і список допустимих адрес призначення повідомлень. Ця інформація зберігається в елементах таблиці процесів, і тому її можна строго контролювати, і нею просто керувати. Інформація про авторизацію ініціюється під час завантаження системи, головним чином, на основі конфігураційних таблиць, створюваних системним адміністратором.

По-четверте, при розробці системи ми явним чином враховуємо можливість до стійкості до деяких збоїв. Всі сервери та драйвери управляються і відслідковуються спеціальним сервером, званим сервером реінкарнації, який може справлятися з двома видами проблем. Якщо системний процес завершується непередбачуваним чином, це негайно розпізнається, і процес перезапускається. Крім того, періодично перевіряється стан кожного системного процесу для перевірки його правильного функціонування. Якщо процес функціонує неправильно, він примусово завершується і перезапускається. Так працює механізм відмовостійкості: зіпсований компонент замінюється, але система весь час продовжує працювати.

**5. Властивості надійності**

Ми вважаємо, що в нашій розробці надійність системи підвищується в порівнянні з усіма іншими існуючими операційними системами за рахунок застосування трьох важливих підходів:

Зменшується кількість критичних збоїв.

Скорочується обсяг шкоди, яка може бути заподіяна будь-який помилкою.

Можна відновити після поширених збоїв.

У наступних підрозділах ми пояснимо, чому застосування цих підходів дозволяє підвищити надійність. Ми також порівняємо вплив деяких класів помилок на нашу систему з тим, як вони впливають на монолітні системи, такі як Windows, Linux і BSD. У розд. 6 ми порівняємо наш підхід до підвищення надійності з іншими ідеями, пропонованими в літературних джерелах.

Скорочення числа помилок в ядрі

Нашої першою лінією захисту є дуже невелике ядро. Добре відомо, що в більшому за обсягом коді міститься більша кількість помилок, і тому чим менше ядро, тим менше в ньому помилок. Якщо в якості нижньої оцінки використати 6 помилок на 1000 рядків виконуваного коду [27], то за наявності 3800 рядків виконуваного коду в ядрі буде присутній, як мінімум, 22 помилки. Крім того, 3800 рядків коду (менше 100 сторінок лістингу, включаючи заголовки та коментарі) – це досить мало, щоб весь цей код міг зрозуміти один чоловік; це істотно підвищує шанси на те, що з часом всі помилки вдасться знайти.

На відміну від цього, в ядрі монолітної системи, такий як Linux, розміром в 2.5 мільйона рядків виконуваного коду, ймовірно, повинно міститися не менше 6 \* 2500 = 15,000 помилок. Крім того, за наявності системи з декількох мільйонів рядків ні одна людина не може прочитати весь вихідний код і повністю зрозуміти, як він працює, що зменшує шанси на знаходження всіх помилок.

Зниження потенційного впливу помилок

Звичайно, зменшення розміру ядра не призводить до скорочення обсягу всього коду системи. При цьому всього лише велика частина системи починає працювати в режимі користувача. Однак саме це зміна надає глибоке вплив на надійність. У коду ядра є можливість повного доступу до всього, що може робити машина. Помилки в ядрі можуть призводити до випадкової ініціалізації введення-виведення, виконання неправильного вводу-виводу, пошкодження таблиць розподілу пам'яті та іншим речам, які не можуть зробити непривілейованих програми, які виконуються в режимі користувача.

Тому ми не стверджуємо, що переведення більшої частини операційної системи в призначений для користувача режим призводить до скорочення загальної кількості наявних помилок. Ми стверджуємо лише те, що ефект прояви помилки при виконанні програми в режимі користувача є менш руйнівним, ніж той, який проявляється при виконанні програми в режимі ядра. Наприклад, аудіо-драйвер, що виконуються в режимі користувача, при спробі використання невірного покажчика насильно завершується сервером процесів, аудіоапаратура перестає працювати, але на іншу частину системи це не впливає.

Для порівняння розглянемо вплив помилки в аудіо-драйвері, що виконуються в режимі ядра. Цей драйвер може ненавмисно перезаписати в стеку адресу повернення зі своєї процедури і зробити при виконанні повернення довільний перехід в монолітне ядро. Цей перехід може привести до коду управління пам'яттю, викликаючи руйнування ключових структур даних, таких як таблиці сторінок і списки вільних і зайнятих ділянок пам'яті. Монолітні системи в цьому відношенні є дуже крихкими і легко руйнуються при прояві помилки.

**Відновлення після збоїв**

Сервери і драйвери запускаються і контролюються системним процесом, званим сервером реінкарнації. Якщо контрольований процес непередбачених чи аварійних чином завершується, це негайно розпізнається, оскільки сервер процесів оповіщає сервер реінкарнації про завершення сервера або драйвера, і процес автоматично перезавантажиться. Крім того, сервер реінкарнації періодично опитує всі сервери і драйвери на предмет їхнього стану. Якщо який-небудь з цих процесів не відповідає правильним чином протягом встановленого інтервалу часу, то сервер реінкарнації насильно завершує і перезапускає погано провідні себе сервери та драйвери. Оскільки дуже багато помилок введення-виведення бувають нестійкими, що проявляються при рідко виникають тимчасових співвідношеннях, синхронізаційних глухий кут і т.д., простий перезапуск драйвера усуває проблему.

Збій драйвера має наслідки і для файлової системи. Можуть бути втрачені невиконані запити вводу-виводу, і в деяких випадках інформація про помилку вводу-виводу доводиться до відома програми. Однак у багатьох випадках можливе повне відновлення. Більш докладне обговорення сервера реінкарнації і надійності на рівні додатків наводиться в розд. 4.

У монолітних системах зазвичай відсутня можливість виявлення збійних драйверів «на льоту», хоча є дані про деякі дослідження в цій області [25]. Тим не менше, заміна на льоту ядерного драйвера є складною справою, оскільки до часу заміни він може утримувати ядерні блокування або знаходитися в критичному ділянці.

Обмеження зловживань переповнювання буфера

Відомо, що переповнення буферів є рясним джерелом помилок, наявністю яких інтенсивно користуються віруси і черв'яки. Хоча наша розробка спрямована радше на боротьбу з помилками, а не із зловмисними кодом, деякі засоби нашої системи надають захист від певних видів зловживань. Оскільки наше ядро є мінімальним, і в ньому використовується тільки статичне розміщення даних, виникнення проблеми малоймовірно в найбільш чутливої частини системи. Якщо переповнення буферу трапляється в одному з користувацьких процесів, то проблема не є надто серйозною, оскільки сервери і драйвери, що виконуються в режимі користувача, володіють обмеженими можливостями.

Крім того, в нашій системі виконується тільки код, розташований в сегментах тексту, які доступні тільки з читання. Хоча це не запобігає можливість переповнення буфера, ускладнюється можливість зловживання, оскільки надлишкові дані, що знаходяться в стеці або купі, неможливо виконати як код. Цей захисний механізм є виключно важливим, оскільки він запобігає зараження вірусами і черв'яками та виконання їх власного коду. Сценарій найгіршого випадку змінюється від взяття безпосереднього управління до перезапису адреси повернення в стеку та виконання деякої існуючої бібліотечної процедури. Найбільш відомий приклад такої ситуації часто називають атакою шляхом «повернення в libc» («return-to-libc»), і цей спосіб атаки вважається набагато більш складним, ніж виконання коду в стеці або купі.

На відміну від цього, в монолітних системах купуються повноваження супер, якщо переповнення буферу відбувається в будь-якій частині операційної системи. Більш того, в багатьох монолітних системах допускається виконання коду в стеці або купі, що істотно спрощує зловживання переповнювання буфера.

Забезпечення надійного IPC

Добре відомою проблемою механізмів обміну повідомленнями є управління буферами, але в нашому варіанті комунікаційних примітивів ми повністю уникаємо цієї проблеми. У нашому механізмі синхронної передачі повідомлень використовуються рандеву, в результаті чого усувається потреба в буферизації і управлінні буферами, а також відсутня проблема вичерпання ресурсів. Якщо одержувач не очікує повідомлення, то примітив SEND блокує відправника. Аналогічно, примітив RECEIVE блокує процес, якщо немає повідомлення, що очікує свого отримання. Це означає, що для заданого процесу в таблиці процесів у будь-який час повинен зберігатися єдиний вказівник на буфер повідомлення.

На додаток до цього, у нас є механізм асинхронної передачі повідомлень NOTIFY, який також не є чутливим до вичерпання ресурсів. Повідомлення є типізовані, і для кожного процесу зберігається тільки один біт для кожного типу. Хоча обсяг інформації, яку можна передати таким чином, обмежений, цей підхід був обраний з-за своєї надійності.

До речі, зауважимо, що у своєму IPC ми уникаємо переповнювання буфера шляхом обмеження засобів комунікації короткими повідомленнями фіксованої довжини. Повідомлення є об'єднанням декількох типізованих форматів повідомлень, так що розмір автоматично вибирається компілятором, як розмір найбільшого допустимого типу повідомлень, який залежить від розміру цілих чисел і покажчиків. Цей механізм передачі повідомлень використовується для всіх запитів і відповідей.

Обмеження IPC

IPC – це потужний механізмом, який потребує строгого контролі. Оскільки наш механізм передачі повідомлень є синхронним, процес, що виконує примітив IPC, блокується, поки обидва учасника не стануть готовими. Користувальницький процес може легко зловживати цим властивістю для завішування системних процесів шляхом посилки запиту без очікування відповіді. Тому є інший примітив IPC SENDREC, що комбінує в одному виклик SEND і RECEIVE. Він блокує відправника до отримання відповіді на запит. З метою захисту операційної системи цей примітив є єдиним, який можна використовувати звичайним користувачам. Насправді, в ядрі для кожного процесу підтримується бітовий масив для обмеження примітивів IPC, які дозволяється використовувати даному процесу.

Крім того, в ядрі підтримується бітовий масив, що визначає, з якими драйверами і серверами може взаємодіяти даний процес. Ця маска посилки повідомлень являє собою механізм, що запобігає безпосередню посилку повідомлень драйверам від користувацьких процесів. Натомість цього, їм дозволяється спілкуватися тільки з серверами, що забезпечують POSIX-дзвінки. Однак маска посилки повідомлень використовується також і для запобігання посилки (непередбаченого) повідомлення, скажімо, від драйвера клавіатури аудіо-драйверу. Знову шляхом суворої інкапсуляції можливостей кожного процесу ми можемо в значній мірі запобігти поширенню неминучих помилок в драйверах і їх вплив на інші частини системи.

На відміну від цього, в монолітній системі будь-який драйвер може викликати будь-який шматок коду в ядрі, використовуючи машинну інструкцію виклику підпрограми (або, ще гірше, інструкцію повернення з підпрограми, якщо стек був перезаписаний через переповнювання буфера), що дозволяє проблем, що виникають в одній підсистемі, поширюватися в інші підсистеми.

Уникання тупиків

Оскільки за замовчуванням для IPC використовуються синхронні виклики SEND і RECEIVE, можуть виникати тупики, коли два або більше число процесів одночасно намагаються обмінюватися повідомленнями, і всі процеси блокуються в очікуванні один одного. Тому ми ретельно розробляли протокол уникнення тупиків, що приписує часткове, що сходить впорядкування повідомлень.

Впорядкування повідомлень приблизно відповідає розбивка на рівні, описаного в розд. 2.2. Наприклад, звичайним користувальницьким процесам дозволяється тільки посилати повідомлення з використанням примітиву SENDREC серверів, які реалізують інтерфейс POSIX, а ці сервери можуть запитувати сервіси від драйверів, які, у свою чергу, можуть виробляти виклики ядра. Однак для асинхронних подій, таких як переривання і таймери, потрібні повідомлення, що посилаються в протилежному напрямку, від ядра сервера або драйверу. Використання синхронних викликів SEND для передачі цих подій може легко призвести до глухого кута. Ми уникаємо цієї проблеми шляхом використання для асинхронних подій механізму NOTIFY, який ніколи не блокує викликає бік. Якщо оповестітельное повідомлення не може бути доставлено процесу-адресату, воно зберігається в його елементі таблиці процесів до тих пір, поки він не виконає RECEIVE.

Хоча протокол уникнення тупиків підтримується обговорювалося вище механізмом масок посилки повідомлень, ми також реалізували в ядрі розпізнавання тупиків. Якщо виклик примітиву в деякому процесі непередбачуваних чином привів би до виникнення безвиході, то виконання примітиву не проводиться, і закликають учасника повертається повідомлення про помилку.

Уніфікація переривань і повідомлень

Базовим механізмом IPC є передача повідомлень на основі рандеву, але потрібні й асинхронні повідомлення, наприклад, для надання інформації про переривання, що є потенційним джерелом помилок в операційних системах. Ми суттєво зменшили тут шанси на появу помилок, уніфікувавши асинхронні сигнали та повідомлення. Зазвичай, коли деякий процес посилає повідомлення іншому процесу і одержувач не є готовим, відправник блокується. Ця схема не працює для переривань, оскільки обробник переривань не може дозволити собі блокування. Замість цього використовується асинхронний механізм сповіщень, при використанні якого обробник переривань виробляє виклик NOTIFY для драйвера. Якщо драйвер очікує повідомлення, то сповіщення доставляється безпосередньо. Якщо він його не очікує, то сповіщення зберігається в бітові масиви до тих пір, поки згодом драйвер не виконає виклик RECEIVE.

Обмеження функціональних можливостей драйвера

Ядро експортує обмежений набір функцій, які можна викликати ззовні. Цей ядерний API представляє собою єдиний спосіб взаємодії драйвера з ядром. Однак не кожному драйверу дозволяється використовувати будь-який виклик ядра. Для кожного драйвера в ядрі (в таблиці процесів) підтримується бітовий масив, який показує, які виклики ядра може виробляти цей драйвер. Гранулярні викликів ядра є досить дрібною. Відсутній мультиплексування викликів в один і той же номер функції. Кожен виклик індивідуально захищається власним бітом в бітові масиви. Проте на внутрішньому рівні кілька викликів може оброблятися однієї і тієї ж ядерної функцією. Цей метод дозволяє реалізувати детальне керування доступом до ядра.

Наприклад, деяким драйверам потрібен доступ по читанню і запису до даних, що знаходяться в призначених для користувача адресних просторах, але виклики для читання і запису в цих просторах є різними. Так що ми не мультіплексіруем читання і запис в один виклик з використанням параметра «напрямок». Відповідно, можна дозволити, наприклад, драйверу принтера виконувати виклик ядра для читання даних з користувацьких процесів, але не дозволяти виконання викликів для запису. Внаслідок цього помилка в драйвері, якому дозволено тільки читання, не може призвести до випадкового пошкодження користувацького адресного простору.

Порівняємо цю ситуацію з можливим поведінкою драйвера у монолітному ядрі. Помилка в коді може призвести до запису в адресний простір користувацького процесу замість читання з нього, що зруйнує процес. Крім того, ядерний драйвер може викликати будь-яку функцію в усьому ядрі, включаючи функції, які не повинні викликатися драйверами. Оскільки відсутня будь-яка внутрішньоядерні захист, це практично неможливо запобігти. У нашій розробці жоден драйвер не може викликати ядерну функцію, яка не була явно експортована як частина інтерфейсу між ядром і цим драйвером.

Заборона доступу до портів введення-виведення

Для кожного драйвера в ядрі підтримується список портів введення-виведення, з яких він може читати, а також тих, у які він може писати. Читання і запис захищаються по окремо, так що процес, у якого є право на тільки читання з деякого порту вводу-виводу, не може писати в нього. Будь-яка спроба порушення цих правил призводить до вироблення коду помилки, що повертається закликають учасника. Таким чином, драйвер принтера може бути обмежений доступом тільки до портів введення-виведення принтера, аудіо-драйвер може бути обмежений доступом тільки до портів введення-виведення звукової карти і т.д.

На відміну від цього, в монолітних системах відсутній спосіб обмеження доступу внутрішньоядерної драйвера тільки до невеликого числа портів введення-виведення. Ядерний драйвер може випадково виконати запис в будь-який порт вводу-виводу і завдати істотної шкоди.

У деяких випадках в адресний простір драйвера можуть відображатися реальні регістри пристрої введення-виведення, щоб уникнути якого б то не було взаємодії з ядром при здійсненні введення-виведення. Однак, оскільки не в усіх архітектурах допускається відображення регістрів введення-виведення в призначені для користувача процеси із забезпеченням необхідного рівня захисту, ми вибрали модель, в якій реальні операції введення-виведення виконуються тільки ядром. Це проектне рішення є ще одним прикладом того, що ми віддаємо перевагу надійності на шкоду ефективності.

Хоча в даний час таблиці, що дозволяють доступ до портів введення-виведення, ініціалізувалися з конфігураційного файлу, ми плануємо реалізувати сервер шини PCI, який буде робити це автоматично. Сервер шини PCI може отримати з BIOS порти введення-виведення, необхідні кожному драйверу, і використовувати цю інформацію для ініціалізації таблиць ядра.

Перевірка параметрів

Оскільки всі виклики ядра проводяться шляхом генерації внутрішнього переривання, ядро може виконати обмежену валідацію параметрів до диспетчеризації дзвінка. Ця валідація включає перевірки як справності (sanity), так і прав доступу (permission). Наприклад, якщо драйвер просить ядро записати блок даних з використанням фізичної адресації, то цей виклик може бути відхилений, оскільки не в усіх драйверів є право на такі дії. Використовуючи віртуальну адресацію, ядро, мабуть, не зможе сказати, чи є ця адреса записи правильним, але воно, принаймні, зможе перевірити, що ця адреса дійсно є допустимим адресою в сегменті даних або стека користувацького процесу, а не відноситься до сегменту тексту і не є якимось випадковим недійсним адресою.

Хоча такі перевірки справності є грубими, це краще, ніж нічого. У монолітних системах ніщо не перешкоджає драйверу виконувати запис за адресами, за якими не можна писати не за яких умов, таким як адреси в сегменті тексту ядра.

Відлов поганих покажчиків

У програмах на мовах C і C + + використовується безліч покажчиків, і ці програми весь час схильні помилок, пов'язаних з використанням поганих покажчиків. Разменованіе невірного покажчика часто призводить до виявлення апаратурою помилки сегментації. У нашій розробці сервер або драйвер, що намагаються разименовать поганий покажчик, примусово завершуються, і видається дамп пам'яті для майбутньої налагодження, точно так само, як і для інших користувальницьких процесів. Якщо поганий покажчик виявляється в частині операційної системи, що виконується в режимі користувача, то сервер реінкарнації негайно помічає наявність збійної ситуації і замінює примусово завершений процес його свіжою копією.

Приборкання нескінченних циклів

Коли драйвер впадає в нескінченний цикл, це створює загрозу споживання нескінченного часу ЦП. Планувальник зауважує наявність такої поведінки і поступово знижує пріоритет несправного процесу, поки він не стає непрацюючим процесом. Проте інші процеси можуть продовжувати нормально працювати. Після вичерпання зумовленого інтервалу часу сервер реінкарняціі помітить, що даний драйвер не відповідає на запити, примусово завершить і перезапустить його.

На відміну від цього, коли в нескінченний цикл впадає ядерний драйвер, він споживає весь час ЦП і фактично завішують всю систему.

Перевірка DMA

Однією з речей, яку ми не можемо забезпечити, є запобігання заподіяння шкоди системі через невірного DMA (Direct Memory Access, прямий доступ до пам'яті). Для запобігання перезапису драйвером через DMA довільної частини реальної пам'яті потрібно апаратна захист. Проте ми можемо виявити деякі помилки DMA наступним чином. DMA зазвичай запускається шляхом запису адреси DMA в деякий порт вводу-виводу. Ми можемо надати бібліотечну процедуру, яка викликається для запису в деякий порт вводу-виводу з попередніми декодуванням (способом, що залежить від пристрою) записів у цей порт вводу-виводу з метою знаходження використовуваних адрес DMA і перевірки їх допустимості. У зловмисних драйверах така перевірка може обходитися, але в добропорядних драйверах цей спосіб дозволяє виловити хоча б деякі помилки при помірних накладних витратах.

Залежно від апаратури ми можемо надійти ще краще. Якщо б на периферійної шині малося MMU (Memory Management Unit, пристрій управління пам'яттю) введення-виведення, ми могли б точно обмежити доступ до пам'яті для кожного драйвера [16]. Для систем з шиною PCI-X ми збираємося покласти на свій сервер шини PCI відповідальність за ініціалізацію таблиць MMU введення-виведення. Це частина нашої майбутньої роботи.

**6. Аналіз надійності**

Для перевірки надійності системи ми вручну внесли деякі ретельно підібрані помилки в деякі з своїх серверів і драйверів, щоб побачити, що в результаті відбудеться. Як описувалося в розд. 3.3, наша система розробляється для виявлення та виправлення багатьох помилок, і саме це ми і спостерігали. Якщо з якої б то не було причини відбувався збій деякого компонента, це розпізнавалася сервером реінкарнації, який застосовував усі необхідні кошти для пожвавлення збійного компонента. Нижче це описується більш детально.

Для розуміння роботи нашої системи потрібно розрізняти два класи помилок. Перший клас складають логічні помилки, що означають, що сервер або драйвер дотримується протоколу межмодульних взаємодій і нормально відповідає на запити, як якщо б він успішно виконав роботу, чого насправді не відбувається. Прикладом є драйвер принтера, який друкує безглузду інформацію, але виробляє нормальні повернення. Для будь-якої системи дуже важко, якщо не неможливо, відловлювати помилки такого роду. Логічні помилки перебувають за межами цього дослідження.

Другий клас складається з протокольних помилок, за наявності яких порушуються правила, що визначають поведінку серверів і драйверів. Наприклад, в нашій системі від серверів і драйверів потрібно відповідати на періодичні запити стану, що надходять від сервера реінкарнації. Якщо вони не підкоряються цьому правилу, робиться коригуючий дію. Наша система розробляється для боротьби з протокольними помилками.

Сервер реінкарнації

Сервер реінкарнації – це центральний сервер, керуючий усіма серверами і драйверами операційної системи. Він дозволяє істотно підвищити надійність, забезпечуючи:

1. Негайне розпізнавання фатальних збоїв.
2. Періодичний моніторинг стану.

Таким чином, він допомагає відловлювати два поширених виду збоїв: померлі або погано себе провідні системні процеси і негайно береться за вирішення найбільш гострої проблеми. Якщо системний процес завершується, то сервер реінкарнації безпосередньо оповіщається про це і перевіряє свої таблиці, щоб зрозуміти, чи слід перезапустити сервіс. Цей механізм, наприклад, забезпечує негайну заміну драйвера, примусово завершеного через використання поганого покажчика. Крім того, періодичний моніторинг стану допомагає дисциплінувати погано себе провідні системні сервіси. Наприклад, драйвер, який впадає в нескінченний цикл і не може відповісти на запит стану від сервера реінкарнації, буде примусово завершений і перезапущений.

Заміна драйвера пристрою складається з суворо контрольованій послідовності дій. По-перше, сервер реінкарнації породжує новий процес, виконання якого затримується, оскільки для нього ще не призначено привілеї. Потім сервер реінкарнації повідомляє про новий драйвері файлової системи і, нарешті, призначає необхідні привілеї. Коли всі ці кроки успішно виконуються, новий процес починає працювати і виконує код драйвера, що береться з файлової системи. В якості додаткової обережності двійковий код деяких драйверів може дублюватися в основній пам'яті, щоб, наприклад, драйвер для диска кореневої файлової системи можна було завантажити без потреби в обміні з диском.

Надійність рівня додатків

Наявність збійного драйвера може приводити до наслідків для файлової системи і додатків, що виробляють введення-виведення. Якщо у файлової системи був невиконаний запит вводу-виводу, їй буде повернуто код помилки, що говорить про збій драйвера. У цей момент можуть бути зроблені різні дії. Необхідно проводити відмінність між блоковими і символьними пристроями, тому що введення-виведення для блокових пристроїв буферізуется в буферному кеші файлової системи. На рис. 3 наводиться огляд різних сценаріїв відновлення на рівні програми.

При фатальному збої блокового драйвера можливо повне відновлення без втрати даних, прозоре для програми. Коли розпізнається збій, сервер реінкарнації запускає нову копію драйвера і скидає кеш файлової системи для синхронізації. Таким чином, буферний кеш не тільки підвищує продуктивність, але також є важливим і для надійності.

Прозоре відновлення іноді є можливим і при збоях драйверів символьних пристроїв. Оскільки запит вводу-виводу не буферізуется в кеші блоків файлової системи, інформація про помилку вводу-виводу повинна бути доведена до програми. Якщо програма не може призвести відновлення, про проблему буде сповіщений користувач. Фактично, збої драйверів проштовхуються нагору, що призводить до різних сценаріїв відновлення. Наприклад, якщо відбувається збій драйвера Ethernet, то мережевий сервер помітить відсутність пакетів і зробить прозоре відновлення, якщо додаток використовує надійний транспортний протокол, такий як TCP. З іншого боку, якщо відбувається збій драйвера принтера, то користувач, звичайно, помітить, що його виведення на друк не вдався і повторить команду друку.

Таким чином, у багатьох випадках наша система може забезпечити повне відновлення на прикладному рівні. В решті випадках інформація про збої введення-виведення доводиться до користувача. Можна було б пом'якшити цю незручність шляхом використання тіньового драйвера для відновлення додатків, який використовували зіпсований драйвер в момент його фатального збою, застосовуючи методи, продемонстровані в [25]. Нам не дає зробити це брак робочої сіли.

Результати перевірки надійності

Для перевірки надійності своєї системи ми вручну внесли збої в деякі з своїх драйверів, щоб протестувати деякі види помилок і подивитися на те, що вийде. У найпростішому випадку ми завершували драйвер з застосуванням сигналу SIGKILL. Більш серйозні тестові випадки змушували драйвери разименовивать погані покажчики або впадати в нескінченний цикл. У всіх випадках сервер реінкарнації розпізнавав проблему і заміняв несправний драйвер свіжої копією.

З тестуванні надійності, ми витягли кілька уроків, важливих для розробки нашої системи. По-перше, оскільки сервер реінкарнації перезапускає несправні сервери і драйвери, потрібно, щоб у них не зберігалося стан, і вони могли б бути належним чином повторно ініціалізували при повторному запуску. Компоненти, що зберігають стан, такі як файлова система і сервер процесів, неможливо вилікувати таким чином, оскільки вони дуже багато втрачають при перезапуску. Наші можливості обмежені.

Інше спостереження полягає в тому, що деякі драйвери були реалізовані таким чином, що ініціалізація відбувається тільки при першому виклику OPEN. Однак для прозорого відновлення після збою драйвера на рівні додатків не повинен турбуватися повторний виклик OPEN. Замість цього, виконання виклику READ або WRITE у відновленому драйвері має змусити драйвер призвести повторну ініціалізацію.

Крім того, хоча ми визнаємо наявність залежностей між файловою системою і драйверами, наші тести виявили деякі інші взаємозалежності. Наприклад, наш інформаційний сервер, що видає на екран налагодження дампи при натисканні функціональних клавіш, втрачає своє відображення клавіш після перезапуску. В якості загального правила, залежності слід запобігати, і всі компоненти повинні бути підготовлені для боротьби з непередбачуваними збоями.

Нарешті, щоб ще більше підвищити надійність, слід змінити і користувальницькі додатки. За історичними причинами в більшості програм передбачається, що будь-який збій драйвера є фатальним, і вони негайно здаються, хоча іноді можливо відновлення. Прикладом, в якому можливе відновлення на рівні програми, є печатка. Якщо демон лінійного принтера сповіщається про тимчасове збій драйвера, він може автоматично повторно видати команду друку без втручання користувача. Подальші експерименти з поновленням на рівні додатків є частиною нашої майбутньої роботи.

**7. Вимірювання продуктивності**

Продуктивність є проблемою, супутньої мінімальним ядер протягом десятиліть. Тому негайно постає питання: у що обходяться що обговорювалися вище зміни? Щоб розібратися в цьому, ми створили прототип, що складається з невеликого ядра і підтримуваного їм набору драйверів пристроїв і серверів, що працюють в режимі користувача. В якості основи прототипу ми почали з використання системи MINIX 2 з-за її невеликого розміру і довгої історії. Код системи вивчався багатьма десятками тисяч студентів в сотнях університетів протягом 18 років, і в останні 10 років майже не надходили повідомлення про помилки, що мають відношення до ядра; мабуть, відсутність помилок пов'язано з малими розмірами ядра. Потім ми значно змінили код, видаливши з ядра драйвери пристроїв і додавши засоби підвищення надійності, що обговорювалися в розд. 3. Таким чином, ми отримали практично нову систему MINIX 3 без потреби у написанні великого обсягу коду, не істотного для даного проекту, такого як драйвери і файлова система.

Оскільки нас цікавить вартість змін, що обговорювалися в даній статті, ми порівнюємо свою систему з базовою системою, в якій драйвери пристроїв є частиною ядра, шляхом запуску одних і тих же тестів на обох системах. Це набагато більш чистий перевірка, ніж порівняння нашої системи з Linux або Windows, яке нагадувало б порівняння яблук з ананасами. Таким порівнянь часто заважають відмінності в якості компіляторів, в стратегіях управління пам'яттю, у файлових системах, в обсязі виконаної оптимізації, в зрілості систем і в багатьох інших факторах, які можуть повністю затінити все інше.

Тестовою системою був 2.2 GHz Athlon (більш точно, AMD64 3200) з 1 Гб основної пам'яті і 40 гігабайтним диском IDE. Жоден з драйверів не був оптимізований для роботи в режимі користувача. Наприклад, ми очікуємо, що на Pentium зможемо забезпечити захищеним чином прямий доступ драйверів пристроїв до необхідних їм портів введення-виведення, усуваючи, таким чином, багато викликів ядра. Однак для підтримки переносимості інтерфейс не буде змінюватися. Крім того, в даний час в драйверах використовується програмований введення-виведення, що набагато повільніше використання DMA. Після реалізації цих оптимізацій ми очікуємо істотного підвищення ефективності. Тим не менше, навіть при використанні існуючої системи погіршення продуктивності виявилося цілком розумним.

Результати тестування системних викликів

Перший пакет тестів містив тести чистих POSIX-сумісних системних викликів. Користувацька програма повинна була зафіксувати реальний час у тактах системних годин (на частоті 60 Гц), потім мільйони раз зробити системний виклик, після чого знову зафіксувати реальний час. Час обробки системного виклику обчислювалося як різниця між кінцевим і початковим часом, поділена на число викликів, за вирахуванням накладних витрат на організацію циклу, які вимірювалися окремо. Число ітерацій циклу було різним для кожного тесту, оскільки тестування 100 мільйонів разів виклику getpid було розумним, але читання 100 мільйонів разів з 64-магабайтного файлу зайняв би надто багато часу. Всі тести виконувалися на незавантажених системі. Для цих тестів частоти успішних звернень до кешу ЦП і кешу файлового сервера імовірно становили 100%.

Коротко проаналізуємо результати цих тестів. Виконання системного виклику getpid зайняло 0.831 мсек при використанні ядерних драйверів і 1.011 мсек при використанні драйверів, що працюють в режимі користувача. При виконанні цього виклику від користувацького процесу менеджеру пам'яті надсилається одиночне повідомлення, на яке негайно виходить відповідь. При використанні драйверів, які виконуються в режимі користувача, виклик виконується повільніше з-за наявності перевірки прав процесів на посилку таких повідомлень. При виконанні такого простого виклику істотне уповільнення викликають навіть кілька додаткових рядків коду. Хоча у відсотках різниця становить 22%, на кожен виклик витрачається лише 180 додаткових наносекунд, так що навіть при частоті 10,000 звернень в секунду втрати складають всього 2.2 мсек в секунду, набагато менше 1%. При виконанні виклику lseek проводиться набагато велика робота, і тому відносні накладні витрати знижуються до 11%. При виконанні відкриття та закриття файлу цей показник становить лише 9%.

Читання і запис 64-кілобайтний ділянок даних займає менше 90 мсек, і падіння продуктивності складає 8%. При використанні драйверів, що виконуються в режимі користувача, створення файлу, запис в нього 1 кілобайт даних і видалення даних займають 13.465 мсек. Через використання буферного кешу файлового сервера в жодному з цих тестів не викликалися драйвери, і тому ми можемо укласти, що інші зміни, не пов'язані з драйверами, сповільнюють систему приблизно на 12%.

Результати тестування дискового введення-виведення

У другому пакеті тестів ми читали з файлу і писали в файл порції від 1 кілобайт до 64 мегабайт. Тести пропускалися багато разів, так що читається файл розміщувався у 12-мегабайтним кеші файлового сервера, крім випадку 64-мегабайтним обмінів, коли обсягу кешу не вистачало. Використання внутрішнього кеша дискового контролера не блокувалося.

Як ми бачимо, різниця в продуктивності становить від 3% до 18%, у середньому – 8.4%. Однак зауважимо, що найгірший показник продуктивності отримано для 1-кілобайтний записів, але абсолютна часом зросла всього на 457 наносекунд. Це співвідношення зменшується при збільшенні обсягу введення-виведення, оскільки скорочуються відносні накладні витрати. У трьох 64-магабайтних тестах, результати яких показані на рис. 6 і 7, це співвідношення становить всього від 3% до 5%.

В іншому тесті проводиться читання з безпосереднього блокового пристрою, відповідного жорсткого диска. Запис на безпосереднє пристрій зруйнувала б його вміст, тому такий тест не виконувався. При виконанні цих тестів не використовується буферний кеш файлової системи, і перевіряється тільки переміщення бітів з диска. Як ми бачимо, в цьому випадку середній показник накладних витрат становить лише 9%.

Результати тестування додатків

Наступний набір тестів складався з реальних програм, а не простих вимірів часу виконання системних викликів. Результати наведено на рис. 8. Перший тест полягав у побудові області початкового завантаження (boot image) у циклі, що містить виклик system («make image»); тим самим, побудова вироблялося багато разів. При кожному побудові компілятор мови C викликався 123 рази, асемблер – 4 рази і компонувальник – 11 разів. Побудова ядра, драйверів, серверів і програми init, а також збірка області початкового завантаження зайняли 3.878 секунд. Середній час компіляції становило 32 мсек на файл.

Другий тест містив цикл, у якому компілювати тести відповідності стандарту POSIX. Набір з 42 тестових програм компілюватися за 1,577 секунди, або приблизно за 37 мсек на файл тесту. Тести з третього по сьомий складалися в сортуванні до 64-мегабайтной файлу та застосування до нього sed, grep, prep і uuencode відповідно. У цих тестах у різних обсягах змішувалися обчислення і обміни з диском. Кожен тест пропускався лише по одному разу, так що кеш файлової системи практично не використовувався, кожен блок брався з диска. Середнє падіння продуктивності склало в цих випадках 6%. Якщо взяти середнє значення для останнього стовпця показників 1922 тестів, відображених на рис. 6–8, ми отримаємо 1.08. Іншими словами, версія з драйверами, що виконуються в режимі користувача, виявилася приблизно на 8% повільніше версії з ядерними драйверами для операцій, які залучають обміни з дисками.

Мережева продуктивність

Ми тестували також і мережеву продуктивність системи з драйверами, що виконуються в режимі користувача. Тестування проводилося з використанням карти Intel Pro/100, оскільки у нас не було драйвера для карти Intel Pro/1000. Ми змогли управляти Ethernet на повній швидкості. Крім того, ми запускали тести поворотної петлі з відправником та одержувачем, що знаходяться на одній машині, і спостерігали пропускну здатність в 1.7 Гб / сек. Оскільки це еквівалентно використанню мережевого з'єднання для посилки на швидкості 1.7 Гб / сек і одночасного прийому на тій же швидкості, ми впевнені, що управління гігабітної апаратурою Ethernet з єдиним односпрямованим потоком на швидкості в 1 Гб / с не повинна створити проблему при використанні драйвера, що виконується в режимі користувача.

Розмір коду

Швидкість – це не єдиний показник, який представляє інтерес; дуже важливим є і кількість помилок. На жаль, ми не можемо безпосередньо перерахувати всі помилки, але розумним замінником числа помилок, ймовірно, є число рядків коду. Нагадаємо: чим більше код, тим більше помилок.

Підрахувати кількість рядків коду не так просто, як може здатися на перший погляд. По-перше, порожні рядки і коментарі не додають в код складності, і тому ми їх не враховуємо. По-друге, # define й інші визначення у файлах заголовків також не додають у код складності, і тому файли заголовків теж не враховуються. Підрахунок числа рядків виконувався з використанням Perl-скрипта sclc.pl, доступного в Internet. Результати для ядра, чотирьох серверів (файлової системи, сервери процесів, сервера реінкарнації, інформаційного сервера), п'яти драйверів (жорсткого диска, флоппі-диска, RAM-диска, терміналу, пристрої журналізацію) і програми init показані на рис. 9.

На малюнку можна бачити, що ядро складається з 2947 рядків на мові C і 778 рядків на мові асемблера (для програмування низькорівневих функціональних можливостей, таких як перехоплення переривань і збереження регістрів ЦП при перемиканні процесів). Всього є 3725 рядків коду. І тільки цей код виконується в режимі ядра. Іншим способом вимірювання розміру коду для C-програм є підрахунок числа точок з комою, оскільки багато операторів мови C завершуються крапкою з комою. У коді ядра є 1729 точок з комою. Нарешті, розмір скомпільованій ядра складає 21,312 байт. Це число задає тільки розмір коду (тобто сегмента тексту). Початкові дані (3800 байт) і стек в це число не входять.

Цікаво, що статистика розмірів коду, показана на рис. 9, представляє мінімальну, але функціонуючу операційну систему. Загальний розмір ядерної частини і частини, що працює в режимі користувача, складає всього 18,000 рядків коду, незвичайно мало для POSIX-сумісної операційної системи.

**8. Споріднені дослідження**

Ми є не першими дослідниками, що намагаються запобігти відмови систем з вини драйверів пристроїв, що містять помилки. І ми не перші намагаємося застосувати мінімальне ядро в якості можливого рішення. Ми навіть не є першими серед тих, що реалізовував драйвери, що працюють в режимі користувача. Тим не менш, ми вважаємо, що ми першими побудували повністю POSIX-сумісну операційну систему з відмінними властивостями ізоляції збоїв поверх мінімального ядра з 3800 рядків; в цій системі кожен драйвер виконується в режимі користувача в окремому процесі, а вся ОС виконується у вигляді декількох призначених для користувача процесів. У цьому розділі ми обговоримо проекти інших дослідницьких груп, які почасти схожі на те, що робимо ми.

Ізоляція драйверів в програмному забезпеченні

Одним з найважливіших дослідницьких проектів, у якому робиться спроба побудувати надійну систему в присутності ненадійних драйверів пристроїв, є Nooks [26]. Метою Nooks є підвищення надійності існуючих операційних систем. Словами авторів, «ми націлюємо існуючі розширення на масові операційні системи, а не пропонуємо нову архітектуру розширень. Ми хочемо, щоб сьогоднішні розширення виконувалися на сьогоднішніх платформах, по можливості, без їх зміни.» Ідея полягає у зворотній сумісності з існуючими системами, але невеликі зміни дозволяються.

Підхід Nooks полягає в тому, щоб залишити драйвери пристроїв у ядрі, але укласти їх у свого роду полегшену захисну оболонку, щоб помилки драйвера не могли поширюватися на інші частини операційної системи. Nooks працює шляхом вставки прозорого рівня підвищення надійності між обертається драйверів пристрою й, що залишився частиною операційної системи. Весь трафік управління і даних між драйвером і залишилася частиною ядра перевіряється рівнем підвищення надійності. При запуску драйвера рівень підвищення надійності модифікує таблицю сторінок ядра таким чином, щоб заборонити доступ по запису до сторінок, які не є частиною драйвера, запобігаючи, тим самим, їхню безпосередню модифікацію. Для підтримки законного доступу по запису в структури даних ядра Nooks копіює необхідні дані в драйвер, а після модифікації переписує їх назад.

Наша мета повністю відрізняється від мети Nooks. Ми не намагаємося зробити більш надійними успадковані системи. Будучи дослідниками, ми задаємо питання: як слід розробляти майбутні операційні системи, щоб із самого початку запобігти виникненню цієї проблеми? Ми вважаємо, що правильна розробка майбутніх систем полягає в побудові мультисерверного операційної системи та виконання ненадійного коду в незалежних процесах в режимі користувача, що зробить цей код набагато менш шкідливим (як обговорювалося в розд. 3).

Незважаючи на різні цілі, є й технічні аспекти, у відношення яких системи можна порівнювати. Розглянемо лише кілька прикладів. Nooks не може впоратися зі складними помилками, такими як ненавмисне зміна в драйвері таблиці сторінок; в нашій системі у драйверів відсутній доступ до таблиці сторінок. Nooks не може впоратися з нескінченними циклами; ми можемо, оскільки, коли драйвер не відповідає правильним чином серверу реінкарнації, він примусово завершується і перезапускається. Хоча на практиці Nooks може в більшості випадків впоратися з неприпустимими записами в структури даних ядра, в нашій розробці такі записи не допускаються структурно. Nooks не може впоратися з драйвером принтера, який випадково намагається зробити запис в порти введення-виведення, керуючі диском; ми відловлюємо 100% таких спроб. Заслуговує на увагу й розмір коду. Nooks включає 22,000 рядків коду, майже в шість разів більше розміру всього нашого ядра і більше мінімальної конфігурації всієї нашої операційної системи. Важко відійти від цієї аксіоми: у більшому за розміром коді міститься більше помилок. Тому статистично Nooks, ймовірно, міститься в п'ять разів більше помилок, ніж у всьому нашому ядрі.

Ізоляція драйверів з використанням віртуальних машин

В іншому проекті з інкапсуляції драйверів це робиться з використанням поняття віртуальної машини для їх ізоляції від інших частин системи [19, 18]. Коли драйвер викликається, він запускається на другий віртуальній машині, не в тій, в якій працює основна система, так що його збій не псує основну систему. Подібно Nooks, цей підхід повністю фокусується на виконанні успадкованих драйверів для успадкованих операційних систем. Автори не стверджують, що для нових розробок хорошим підходом є включення ненадійного коду в ядро з подальшою захистом кожного драйвера шляхом його виконання на окремій віртуальній машині.

Хоча цей підхід дозволяє досягти намічених цілей, з ним пов'язані деякі проблеми. По-перше, є питання, пов'язані з тим, наскільки можуть довіряти один одному основна система та віртуальна машина, на якій виконується драйвер. По-друге, запуск драйвера на віртуальній машині породжує проблеми з тимчасовими співвідношеннями і блокуваннями, оскільки всі віртуальні машини працюють у режимі поділу часу, і ядерний драйвер, що розроблявся в розрахунку на виконання без переривань, може бути непередбачуваним чином квантованих в часі з непередбачуваними наслідками. По-третє, може знадобитися спільне використання кількома віртуальними машинами деяких ресурсів, таких як конфігураційне простір шини PCI. По-четверте, механізм віртуальної машини споживає додаткові ресурси, хоча відповідні витрати сумірні з витратами нашої схеми: від 3% до 8%. Хоча для цих проблем пропонуються рішення, підхід у кращому випадку є громіздким і в основному підходить для захисту успадкованих драйверів в успадкованих операційних системах, а не для використання в нових розробках, яким присвячено наше дослідження.

Засоби безпеки, засновані на мовах

У попередній роботі один з авторів також торкався проблему безпечного виконання зовнішнього коду всередині ядра. У проекті Open Kernel Environment (OKE) забезпечується безпечна, що контролює ресурси середовище, що дозволяє завантажити в ядро операційної системи Linux повністю оптимізований власний код [4]. Код компілюється з використанням спеціального компілятора Cyclone, який додає до об'єктному коду інструментарій у відповідності з політикою, яка визначається привілеями користувача. Cyclone, подібно Java, є мовою з типовою безпекою, в якому більша частина помилок, пов'язаних з покажчиками, запобігається мовними засобами. Явне довірче управління (trust management) і контроль авторизації забезпечують адміністраторам можливість здійснювати суворий контроль над наданням зовнішнім модулям привілеїв, і цей контроль автоматично приводиться у виконання в коді цих модулів. Крім забезпечення авторизації, компілятор грає центральну роль в перевірці того, що код відповідає встановленої політиці. Для цього використовуються як статичні перевірки, так і динамічний інструментарій.

OKE дозволяє зовнішнім модулям інтенсивно взаємодіяти з іншими частинами ядра, наприклад, шляхом спільного використання пам'яті ядра. Робоча середовище забезпечує ключові засоби безпеки. Зокрема, для даних завжди проводиться прибирання сміття, і не може відбутися звернення за вказівником до вільної пам'яті. Більш того, OKE може забезпечувати контроль над усіма ресурсами зовнішніх модулів ядра: час ЦП, купа, стек, точки входу і т.д.

Середа OKE розроблялася в розрахунку на написання драйверів і розширень ядра. Проте, оскільки для забезпечення безпечного програмування в ядрі Linux потрібні процедури суворого контролю доступу і складні засоби, середу досить важко використовувати. Як відзначають автори, основна причина полягає в тому, що організація Linux просто не призначена для забезпечення можливості безпечних розширень.

Віртуальні машини і екзоядра

Класичні віртуальні машини [24] представляють собою потужний засіб для одночасного виконання кількох операційних систем. Екзоядра [10] схожі на віртуальні машини, але в них ресурси швидше розділяються, а не реплікуються, що призводить до більшої ефективності. Проте жоден з цих підходів не вирішує проблему, поставлену в розд. 1.3: як запобігти відмови операційних систем з вини драйверів пристроїв, що містять помилки?

Драйвери, що виконуються в режимі користувача в монолітному ядрі

Раннім проектом, в якому застосовувалися драйвери, що виконуються в режимі користувача, був Mach 3.0 [11]. Система складалася з мікроядра Mach, поверх якого запускалася ОС Berkeley UNIX у вигляді користувацького процесу, і драйвери пристроїв також виконувалися в призначених для користувача процесах. На жаль, у разі фатального збою драйвера Berkeley UNIX доводилося перезапускати, так що від ізоляції драйверів було мало користі. Планувалася мультисерверного система, яка повинна була виконуватися над Mach, але вона так і не була повністю реалізована.

В аналогічному проекті в університеті New South Wales реалізовувалися драйвери Linux для жорсткого диска і гігабайтної апаратури Ethernet, що виконуються в режимі користувача [8]. Для блоків розміром менше 32 Кб продуктивність ядерного драйвера була значно вище, але на блоках більшого розміру вирівнювався. Під час тестування Ethernet виявилося так багато аномалій, ймовірно, пов'язаних з управлінням буферами, але не можна було зробити які-небудь висновки.

Розробки мінімальних ядер

Хоча витяг драйверів з ядра є великим кроком вперед, ще краще витягти з ядра операційну систему. Саме тут починають застосовуватися мінімальні ядра з надзвичайних скороченням числа реалізованих у них абстракцій. Ймовірно, першим мінімальним ядром була система RC4000 Брінка Хансена (Brinch Hansen), що датується початком 1970-х рр. [13]. З середини 1980-х рр. був написаний ряд мінімальних ядер, включаючи Amoeba [21], Chorus [5], Mach [1] і V [6]. Проте ні в одному з них не застосовувалося безпечне програмне забезпечення: у всіх було не ізольовані драйвери всередині ядра.

QNX є комерційною UNIX-подібної системою реального часу з закритими кодами [17]. Хоча у неї є мінімальне ядро, зване Neutrino, з приводу системи опубліковано мало статей, і точні деталі нам невідомі. Проте на основі останніх проспектів ми робимо висновок, що Neutrino є гібридним ядром, оскільки менеджер процесів працює в адресному просторі ядра.

На початку 1990 рр. покійний Йохан Лідтке (Jochen Liedtke) написав мінімальне ядро L4 мовою асемблера для архітектури x86. Швидко стало зрозуміло, що воно не є стерпним, і його важко підтримувати, і тому він переписав ядро на мові C [20]. Після цього воно продовжувало розвиватися. В даний час є дві основні гілки: L4/Fiasco, підтримуване в технічному університеті Дрездена, і L4Ka: Pistachio, підтримуване в університеті Карлсруе та університеті New South Wales. Вони написані на C + +.

Ключовими ідеями в L4 є адресні простору, нитки і IPC між нитками в різних адресних просторах. Менеджер ресурсів, що виконується в режимі користувача та запускається при завантаженні системи, управляє системними ресурсами і розподіляє їх між користувацькими процесами. L4 – це одне з небагатьох дійсно мінімальних ядер з драйверами пристроїв, що працюють у режимі користувача. Проте відсутня реалізація, в якій кожен драйвер виконувався б в окремому адресному просторі, і API L4 зовсім відрізняється від нашого API, тому ми не можемо запустити на ньому будь-які тести.

Однак виявилося неважко запустити скрипт підрахунку числа рядків над поточною версією ядра L4Ka: Pistachio. Результати показані на рис. 10, і їх можна порівняти з даними на рядку «Kernal». Розмір початкового коду майже у два рази перевищує розмір нашого ядра, а бінарний код у шість разів більше, проте функціональні можливості L4Ka: Pistachio є зовсім іншими, так що важко сказати що-небудь ще, крім того, що це ядро значно більше за розміром.

Односерверні операційні системи

Одним зі способів використання мінімальних ядер є забезпечення платформи, поверх якої, як єдиний сервер, запускається вся операційна система, можливо, в режимі користувача. Для отримання системних сервісів для користувача програми запитують їх у процесу операційної системи. Властивості такої архітектури аналогічні властивостям монолітних систем, що обговорювалося в розд. 2.1. Помилка в драйвері як і раніше може зламати всю операційну систему, а в результаті і прикладні програми. Тому, з точки зору ізоляції збоїв, виконання всієї операційної системи в одному користувача процесі нітрохи не краще її виконання в режимі ядра. Єдиним реальним перевагою є те, що перезавантаження після фатального збою сервера операційної системи, виконується в режимі користувача, і всіх додатків відбувається швидше, ніж перезавантаження комп'ютера.

Одним із прикладів цієї технології є ОС Berkeley UNIX поверх Mach (перейменована в Darwin компанією Apple), яка є основою системи Apple Mac OS X [28]. Однак у цій системі UNIX виконується в ядрі, що робить його просто інакше структурованим монолітним ядром. Другий приклад – ОС MkLinux, в якій Linux виконується в єдиному користувача процесі поверх Mach. Третій приклад – L4-Linux, в якій повний варіант Linux виконується поверх L4 [15]. В останній з перерахованих систем користувальницькі процеси отримують сервіси операційної системи шляхом виклику віддалених процедур у сервері Linux з використанням механізму IPC L4. Вимірювання показують падіння продуктивності в порівнянні із звичайною ОС Linux на 5–10%, що дуже близько до нашими спостереженнями. Однак єдина рядок з помилковим кодом в драйвері Linux може призвести до збою фатального всієї операційної системи, так що єдиним перевагою цієї архітектури з точки зору надійності є більш швидке завантаження.

Мультисерверного операційні системи

Більш складний підхід полягає в розщепленні операційної системи на частини і виконання кожної частини у власній області захисту. Одним з таких проектів був SawMill Linux [12]. Проте в 2001 р. проект був несподівано зупинений після того, як багато хто з його основних учасників пішли з IBM.

Іншим мультисерверного проектом є DROPS, в якому ОС також будується поверх мінімального ядра L4/Fiasco [14]. Цей проект орієнтована на мультимедійні додатки. Однак більшість драйверів пристроїв виконується у складі великого серверного процесу L4-Linux, і тільки мультимедійні підсистеми виконуються окремо. Після деякої настройки програш в продуктивності знизився до 2–4%.

Ще однією мультисерверного операційною системою з драйверами, що виконуються в режимі користувача, є Nemesis [23]. У цій системі є єдиний адресний простір, розділяється всіма процесами, але використовується апаратна захист між процесами. Подібно DROPS ця система була орієнтована на мультимедійні додатки, але не була POSIX-сумісною і навіть UNIX-подібної.

**Висновок**

Основне досягнення роботи, описаної в цій статті, полягає в тому, що ми побудували POSIX-сумісну операційну систему, засновану на мінімальному ядрі, вихідні тексти якого складають менше 3800 рядків. Тільки цей код виконується в режимі ядра. Наскільки нам відомо, наше мінімальне ядро є найменшим серед усіх існуючих ядер, які підтримують повністю POSIX-сумісну мультисерверного операційну систему, яка функціонує у режимі користувача. Унікальність нашої системи полягає також у тому, що в ній кожен драйвер пристрою виконується в окремому користувача процесі, і є можливість реінкарнації бездіяльних або невірно функціонуючих драйверів на льоту, без перезавантаження операційної системи. Ми не стверджуємо, що можемо відловити будь-яку помилку, але ми істотно підвищили надійність операційної системи шляхом структурного усунення багатьох різних класів помилок.

Для досягнення максимальної надійності у своїй розробці ми керувалися принципами простоти, модульності, найменшою авторизації і відмовостійкості. У розуміється і мінімальному ядрі міститься менша кількість помилок, і воно в меншій мірі піддається фатальним збоїв. Наприклад, у нашому коді ядра неможливі переповнення буферів, оскільки всі структури даних у ньому оголошуються статично, а не з використанням динамічного розподілу пам'яті. Крім того, шляхом переміщення більшої частини коду (і більшої частини помилок) у непривілейованих користувальницькі процеси і обмеження можливостей кожного з них ми домоглися належної ізоляції збоїв і обмежили масштаб відповідного потенційного збитку. Більш того, більшість серверів і всі драйвери в операційній системі піддаються моніторингу і автоматично відновлюються при виявленні проблеми. За це скорочення числа фатальних збоїв операційної системи ми платимо зниженням продуктивності на 5–10%. Ми вважаємо цю ціну цілком обгрунтованою.

Звичайно, драйвери, файлові системи та інші компоненти не стають в нашій розробці магічним чином безпомилковим. Однак за наявності стабільного мінімального ядра сценарій найгіршого випадку змінюється від потреби в перезавантаженні комп'ютера до потреби в перезапуску операційної системи в режимі користувача. Принаймні, це відновлення відбувається набагато швидше. У кращому випадку, якщо, скажімо, в драйвері принтера виникає аварійний відмова з причини записи по невірному вказівником, сервер реінкарнації автоматично запускає свіжу копію цього драйвера. Потрібно заново виконати поточне завдання на друк, але все це ніяк не вплине на інші програми, які виконувалися до моменту фатального збою драйвера. Ситуація з блоковими пристроями справи ще краще. Якщо виявляється збій дискового драйвера, то система може зробити повне відновлення шляхом прозорої заміни драйвера і перезапису блоків з буферного кешу файлової системи.

На завершення статті зазначимо, що ми показали, як можна підвищити надійність операційної системи з використанням елегантного, полегшеного підходу. Наша система в цей час є стійкою до більшості видів невірної роботи, що викликається помилками. Проте є нові проблеми, пов'язані із зловмисними серверами і драйверами. Ми продовжуємо дослідницьку роботу в цій області.