Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jan Rękorajski
Nr albumu: 243017

Urządzenie blokowe do symulowania awarii

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2008

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

W pracy przedstawiam implementację sterownika urządzenia blokowego umożliwiającego symulowanie różnego rodzaju awarii. Jest on przeznaczony dla systemu Linux. Pozwala na testowanie zachowania systemów plików, urządzeń logicznych i oprogramowania użytkowego korzystającego bezpośrednio z urządzeń blokowych w przypadku wystąpienia błędów wejścia-wyjścia. Omawiam również awarie nośników, ich przyczyny i skutki.

Słowa kluczowe

urządzenie blokowe, awaria, sterownik, Linux, mapowanie urządzeń

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.4 Operating Systems
D.4.2 Storage Management
D.4.5 Reliability

Tytuł pracy w języku angielskim

Fault simulating block device

Spis treści

Wprowadzenie . . . 7

1. Urządzenia do przechowywania danych i ich awaryjność . . . 9
1.1. Nośniki . . . 9
1.1.1. Dyski optyczne . . . 9
1.1.2. Dyski magneto-optyczne . . . 9
1.1.3. Dyski magnetyczne . . . 9
1.1.4. Pamięć flash . . . 10
1.1.5. SSD . . . 10
1.2. Rodzaje awarii . . . 10
1.3. Marketing a rzeczywistość . . . 11

2. Mechanizmy symulowania awarii . . . 15
2.1. Elementy podsystemu blokowego . . . 15
2.2. Śledzenie pracy podsystemu blokowego . . . 16
2.3. Symulatory dysków . . . 17
2.4. Sterownik *faulty* podsystemu MD w Linuksie . . . 17
2.4.1. Podsystem MD . . . 17
2.4.2. Sterownik *faulty* . . . 18
2.5. Rozwiązania specjalizowane . . . 18

3. Projekt i implementacja sterownika . . . 21
3.1. Założenia . . . 21
3.1.1. Podsystem blokowy w Linuksie . . . 21
3.1.2. *Device Mapper* . . . 22
3.1.3. Interfejs podsystemu *Device Mapper* . . . 22
3.2. Sterownik dm-fault . . . 25
3.2.1. Implementacja . . . 25
3.2.2. Struktury danych . . . 26
3.2.3. Tryby pracy sterownika . . . 28
3.3. Biblioteka libdmfault . . . 29

4. Testy . . . 31
4.1. Testy poprawności . . . 31
4.2. Testy transakcyjności systemów plików . . . 32
4.2.1. Wyniki . . . 33

5. Podsumowanie . . . 37

A. Zawartość płyty dołączonej do pracy 39

Bibliografia 41

Spis rysunków

1.1. Wzorzec awaryjności dysków 12
1.2. Procentowy udział błędów zgłoszonych przez S.M.A.R.T. w ogólnej liczbie uszkodzonych dysków [PWB07] 13

2.1. Uproszczony schemat podsystemu blokowego 15

4.1. Liczba operacji niewidocznych po awarii 34
4.2. Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync* 34
4.3. Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *dirsync* 35
4.4. Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync* i *dirsync* 35
4.5. Liczba operacji niewidocznych po awarii i uszkodzeniu ostatniego zapisu . . . 35
4.6. Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync* i uszkodzeniu ostatniego zapisu 36
4.7. Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *dirsync* i uszkodzeniu ostatniego zapisu 36
4.8. Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync* i *dirsync* i uszkodzeniu ostatniego zapisu 36

Wprowadzenie

W obecnych czasach wraz z gwałtownie rosnącą pojemnością urządzeń pamięci masowych, coraz większym problemem stają się ich awarie. Dotyczy to zarówno indywidualnych użytkowników, jak i dużych centrów danych.

Każda awaria wiąże się z częściową lub całkowitą utratą danych i koniecznością ich odtworzenia z kopii zapasowej. Niestety, każda taka operacja powoduje krótszy lub dłuższy przestój w działaniu systemu, a w skrajnych przypadkach, gdy utraconych danych nie ma w kopii zapasowej, należy je utworzyć od nowa. Wbrew pozorom awaria może być bardziej dotkliwa dla zwykłego użytkownika, ponieważ często nie ma on świadomości jak bardzo skomplikowanymi i przez to podatnymi na uszkodzenia są nośniki danych. Innym powodem jest koszt zabezpieczenia danych, który może być zbyt duży dla użytkownika, natomiast dla centrum danych jest niewielki w porównaniu z kosztem ich odtworzenia.

Rozwój technologii przechowywania danych i związana z tym różnorodność nośników powodują, że rośnie liczba elementów mogących ulec uszkodzeniu, począwszy od sterowników w systemie operacyjnym, poprzez pamięć komputera, kontrolery obsługujące urządzenia pamięci masowych, przewody łączące elementy ze sobą, a na oprogramowaniu wbudowanym w urządzenia skończywszy.

W pewnych przypadkach może się zdarzyć, że niegroźna z pozoru awaria, np. odłączenie zasilania, powoduje dużo poważniejsze szkody z powodu błędów w oprogramowaniu systemowym odpowiedzialnym za obsługę danych. Oprogramowanie to to nie tylko sterowniki urządzeń i warstwa jądra zajmująca się ich obsługą, lecz także warstwy wyższe, takie jak sterowniki urządzeń logicznych RAID (ang. *Redundant Array of Inexpensive Disks*), menedżery woluminów, czy systemy plików.

W tej sytuacji istotne wydaje się testowanie oprogramowania systemowego pod względem odporności na awarie nośników. Niestety istniejące rozwiązania, takie jak symulatory dysków i programy śledzące, pozwalają przede wszystkim badać wydajność pamięci masowych i ich obsługę przez system operacyjny, natomiast nie udostępniają wygodnych mechanizmów pozwalających na generowanie awarii. Powstają też specjalizowane sterowniki, stosowane w celu wykonania symulacji o bardzo wąskim zakresie lub tak ogólne, że nie umożliwiają pełnej kontroli nad zachowaniem urządzenia. Stąd też powstał pomysł napisania sterownika umożliwiającego symulację awarii w konfigurowalny i kontrolowany sposób.

W pracy zaprezentuję sterownik urządzenia blokowego dla systemu Linux pozwalający na symulowanie różnego rodzaju awarii, takich jak:

- błędy odczytu i zapisu,
- uszkodzenie bloków danych.

Ważną cechą sterownika jest jego duża elastyczność pozwalająca na tworzenie skomplikowanych scenariuszy awarii. Dzięki wykorzystaniu standardowych mechanizmów systemu Linux do budowy i kontroli sterownika, możliwe jest zastosowanie go do testowania nie tylko

oprogramowania systemowego, ale też użytkowego, na przykład bazy danych korzystającej bezpośrednio z urządzeń blokowych.

W rozdziale 1 opiszę rodzaje nośników, pokażę że pomimo zapewnień producentów awarie dysków są cały czas istotnym problemem, a także opiszę najczęstsze typy awarii. W rozdziale 2 opiszę dostępne mechanizmy badania pracy dysków i symulowania awarii. Rozdział 3 poświęcony będzie projektowi i implementacji nowego sterownika. Rozdział 4 zawiera wyniki testów. Rozdział 5 jest podsumowaniem pracy. Dodatek A opisuje zawartość płyty.

Rozdział 1

Urządzenia do przechowywania danych i ich awaryjność

1.1. Nośniki

Pisząc nośniki danych najczęściej mamy przed oczami dyski magnetyczne. Są one najpopularniejszym, ale nie jedynym medium używanym do przechowywania danych. W kolejnych punktach przedstawię kilka najpopularniejszych nośników.

1.1.1. Dyski optyczne

Dyski optyczne, takie jak CDROM (ang. *Compact Disc Read-Only Memory*), DVD (ang. *Digital Versatile Disc*), czy najnowszy Blu-Ray działają na zasadzie odbicia światła laserowego od specjalnego materiału (najczęściej jest to aluminium), w którym dane są wytłoczone, a w przypadku dysków wielokrotnego zapisu wypalone, w reprezentacji bitowej. Pojemność dysku optycznego zależy od jego średnicy i długości fali świetlnej użytej w laserze. W przypadku CDROM jest to laser podczerwony, pozwalający zmieścić 700 MB danych ma płycie, dla DVD laser czerwony dzięki czemu pojemność wzrosła do 4.7 GB, najnowsze dyski Blu-Ray wykorzystują laser niebiesko-fioletowy, co pozwoliło na zwiększenie pojemności do 25 GB.

1.1.2. Dyski magneto-optyczne

Dyski magneto-optyczne wykorzystują laser do odczytu danych ze specjalnego nośnika magnetycznego. W zależności od namagnesowania nośnika zmienia się długość odbitego światła. Aby zapisać dane laser podgrzewa nośnik, a specjalny elektromagnes zmienia jego polaryzację w danym miejscu. Ponieważ napęd magneto-optyczny przy każdym zapisie kontroluje poprawność i od razu zgłasza do systemu problemy, dyski magneto-optyczne są pewniejsze w użyciu niż dyski optyczne, w których takie testy nie są wykonywane. Dyski magneto-optyczne występują w pojemnościach od 128 MB do 2.3 GB.

1.1.3. Dyski magnetyczne

Dyski magnetyczne jako nośnik danych wykorzystują materiał ferromagnetyczny, na którym dane zapisywane są poprzez namagnesowanie odpowiednich fragmentów. W zależności od polaryzacji namagnesowania specjalna głowica odczytuje 0 lub 1. Najpopularniejszym zastosowaniem dysków magnetycznych są dyski twarde obecnie wykorzystywane praktycznie we

wszystkich komputerach. Nośniki magnetyczne używane były również w dyskietkach komputerowych, które z powodu swojej nietrwałości oraz małej odporności na czynniki zewnętrzne i uszkodzenia zostały wyparte przez pamięci flash.

1.1.4. Pamięć flash

Pamięć flash jest rodzajem nieulotnej pamięci komputerowej, która może być elektronicznie kasowana i zapisywana. Używana jest w kartach pamięci i dyskach USB-flash. Flash to specjalny typ pamięci EEPROM (ang. *Electrically Erasable Programmable Read-Only Memory*), która może być zapisywana w dużych blokach dzięki czemu zapis jest dużo szybszy niż w standardowej pamięci EEPROM. Pamięci flash dzięki swoim niewielkim rozmiarom, wygodzie użytkowania i bardzo dużej odporności na uszkodzenia mechaniczne całkowicie wyparły z użycia przenośne nośniki magnetyczne.

1.1.5. SSD

SSD (ang. *Solid State Drive, Solid State Disk*) to urządzenie, które do przechowywania danych wykorzystuje pamięć RAM (ang. *Random Access Memory*) lub opisaną wyżej pamięć flash. SSD tym różni się od typowych pamięci flash, że w pełni emuluje dysk twardy dzięki czemu łatwo może taki dysk zastąpić. W przypadku dysków SSD opartych na ulotnej pamięci RAM dysk taki zawiera baterię zapobiegającą utracie danych.

1.2. Rodzaje awarii

Możemy wyróżnić kilka rodzajów awarii w zależności od przyczyn i skutków jakie mogą powodować. Mówiąc o awarii urządzeń pamięci masowych, na ogół myślimy, że uszkodzeniu uległ nośnik. Jak się okazuje, rzeczywistość jest nieco inna. Badania opisane w pracy [JiHuZ08] pokazują że dyski są odpowiedzialne za 20-55% awarii pamięci masowych, 27-68% jest spowodowane problemami ze sprzętem obsługującym nośniki, zaś błędy w oprogramowaniu sterującym pracą urządzeń odpowiadają za 5-10% awarii.

Najczęstsze przyczyny awarii to:

- uszkodzenie nośnika — powodowane zużyciem, jak również wpływem czynników zewnętrznych, takich jak kurz. Głowica dysku porusza się w odległości nanometrów od talerza, więc jeśli cokolwiek dostanie się do wnętrza mechanizmu może spowodować zarysowanie powierzchni i zniszczenie nośnika. W przypadku dysków optycznych drobne zarysowanie powierzchni ma wpływ na błędny odczyt danych przez laser.

- uszkodzenie części mechanicznych — dotyczy przede wszystkim dysków magnetycznych, które są skomplikowanymi mechanizmami. Miałem do czynienia z dyskiem, w którym zakleszczeniu uległ silnik sterujący pracą głowic do odczytu i zapisu danych.

- uszkodzenie elektroniki — w przypadku nośników takich jak pamięć flash, elektronika jest właściwie nośnikiem danych, natomiast w przypadku dysków magnetycznych, służy do kontroli pracy mechanizmu.

- awaria zasilania — nie powoduje trwałych uszkodzeń, jednak może mieć wpływ na dane zawarte na nośniku jeśli nastąpi w czasie wykonywania operacji zapisu.

- przekłamania danych spowodowane wadliwym okablowaniem — niejednokrotnie zdarza się, że dysk, który w warunkach testowych zostanie uznany za sprawny, nie działa w

systemie docelowym, gdyż wadliwe kable powodują, że do dysku dociera zakłamana informacja, przez co sprawia on wrażenie uszkodzonego.

- błędy w oprogramowaniu — po pierwsze mogą zdarzyć się błędy w sterownikach systemu operacyjnego, od najprostszych, powodujących, że system nie jest w stanie obsłużyć urządzenia, do bardziej skomplikowanych, których efektem mogą być przekłamania danych. Po drugie, niektóre nośniki i większość sprzętu do ich obsługi zawierają oprogramowanie sterujące, które również może posiadać błędy, w skrajnych przypadkach powodujące zniszczenie danych zawartych na nośniku.

Wpływ awarii na funkcjonowanie systemu:

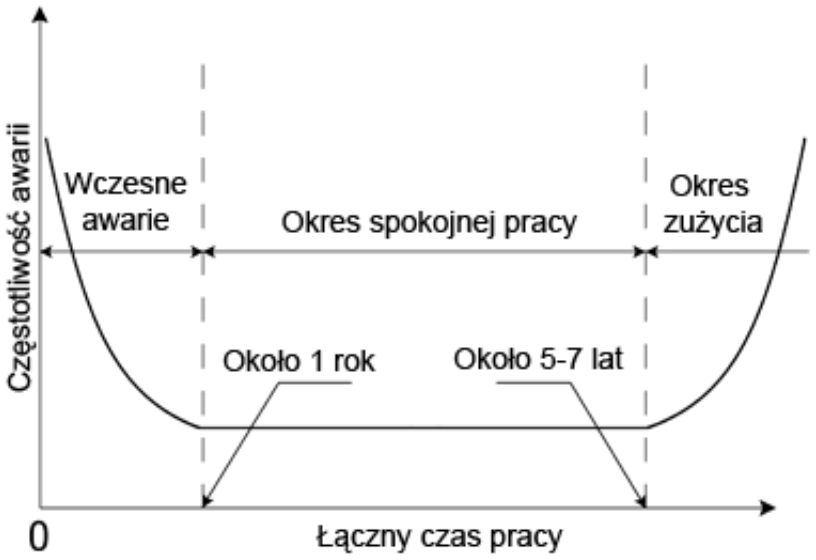
- nieszkodliwe — w przypadku użycia rozwiązań zapewniających redundancję danych, takich jak macierze RAID, awaria jednego lub więcej dysków może nie naruszać ciągłości pracy systemu i bezpieczeństwa danych.
- niska szkodliwość — jeśli uszkodzenie nie jest trwałe i uda się je wykryć odpowiednio wcześnie, to jest duża szansa na uniknięcie strat.
- duża — aby spowodować poważne problemy nie jest konieczna całkowita awaria dysku uniemożliwiająca dalszą pracę, wystarczy, że uszkodzeniu ulegną partie nośnika wrażliwe dla funkcjonowania systemu plików. W takiej sytuacji, pomimo że dane znajdują się na nośniku, ich odzyskanie jest bardzo trudne, a w najgorszym przypadku niemożliwe.

1.3. Marketing a rzeczywistość

Wraz z rozwojem technologii producenci dysków zapewniają użytkowników, że ich produkty są coraz mniej awaryjne. W specyfikacjach dysków widać coraz większe liczby opisujące czas w jakim może wystąpić uszkodzenie, 1 000 000 do 1 500 000 godzin do awarii (MTTF, ang. *Mean Time To Failure*) jest obecnie standardem. Wydawać by się mogło, że prawdopodobieństwo uszkodzenia jest tak nikłe, że użytkownicy nie powinni się tym przejmować. Badania przeprowadzone w ostatnich latach pokazują jednak dużą rozbieżność między deklaracjami producentów a rzeczywistością.

Na konferencji USENIX FAST'07 (*5th USENIX Conference on File and Storage Technologies*) w roku 2007 przedstawiono dwa opracowania dotyczące awaryjności dysków na przestrzeni kilku lat. W pierwszym z nich (por. [SG07]) naukowcy z uniwersytetu Carnegie Mellon zaprezentowali analizę danych dotyczących wymiany dysków w kilku dużych centrach danych. Według specyfikacji producentów przy czasach od 1 000 000 do 1 500 000 godzin do awarii sugerowany roczny wskaźnik awarii (AFR, ang. *Annual Failure Rate*) powinien wynosić co najwyżej 0.88%. Jak się okazało w czasie badań typowy wskaźnik awarii wynosi ponad 1%, przy czym normą jest 2-4%, a w niektórych przypadkach osiągał on nawet 13%. Ponadto w badaniach pominięto przypadki błędnych partii czyli dysków, które ulegały uszkodzeniom bardzo szybko z powodu nieprawidłowości w czasie produkcji. Dodatkowo okazało się, że liczba awarii nie jest stała w czasie i nie zgadza się z powszechnie przyjmowanym dla dysków tzw. wykresem wanny (por. rys. 1.1).

Według niego dyski ulegają awariom w ciągu pierwszych kilku miesięcy, następnie przez 4-6 lat ich awaryjność jest bardzo mała, by później gwałtownie wzrosnąć z powodu zużycia. Z obserwacji wynika, że wczesne awarie nie są takim dużym problemem jak mogłoby się wydawać i nie ma czegoś takiego jak okres spokojnej pracy, w czasie którego można się spodziewać że dysk awarii nie ulegnie. Wyraźnie widać natomiast, że dyski ulegają ciągłemu



Rysunek 1.1: Wzorzec awaryjności dysków

zużyciu i liczba uszkodzeń rośnie stale w czasie ich pracy. Inną interesującą obserwacją było to, że awaryjność dysków nie zależy od technologii w jakiej zostały wykonane, rozkłady awarii były bardzo podobne dla dysków SCSI, FC czy SATA. Wynika z tego, że większy wpływ na awaryjność mogą mieć czynniki niezależne od dysków, takie jak warunki pracy, czy inne elementy systemu (np. okablowanie, kontrolery). Jak widać z powyższego opracowania faktyczna awaryjność dysków znacznie różni się od przewidywanej przez producentów i awaryjność nie jest zależna od technologii w jakiej wykonane są dyski.

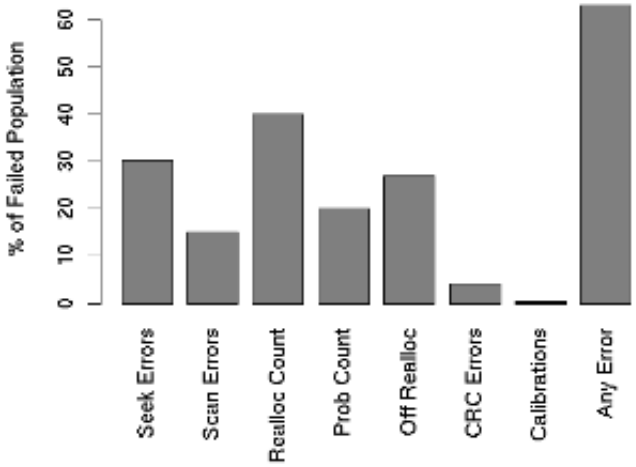
Drugim opracowaniem przedstawionym na konferencji FAST'07 była analiza danych zebranych z wewnętrznego monitoringu dysków we wszystkich systemach działających w firmie Google [PWB07]. Do badań wykorzystano system S.M.A.R.T. (ang. *Self-Monitoring Analysis And Reporting Technology* obecny w nowoczesnych dyskach. System ten umożliwia monitorowanie pracy dysku, naprawianie pewnych typów błędów oraz informowanie użytkownika o potencjalnych problemach. Z obserwacji wynika, że wbrew powszechnemu mniemaniu, intensywność pracy czy temperatura ma niewielki wpływ na awaryjność, a wręcz wpływ ten może być zupełnie inny niż się uważa. Okazało się, na przykład, że wyższa temperatura może lepiej wpływać na dysk niż niższa. Jest to związane z tym, że producenci testując dyski w ekstremalnych warunkach spowodowali, że lepiej działają przy temperaturach wyższych niż zalecane. Opracowanie pokazuje, że część informacji uzyskiwanych z systemu S.M.A.R.T. jest bardzo dobrym sygnałem zbliżającej się awarii. Jednak w bardzo wielu przypadkach system ten nie daje niestety żadnych wcześniejszych sygnałów co powoduje, że nie nadaje się do wykorzystania w tworzeniu modeli przewidywania awarii. Widać to bardzo wyraźnie na rysunku 1.2, aż 56% dysków nie wykazywało najbardziej znaczących sygnałów, którymi są:

- błędy skanowania (ang. *scan errors*) — napędy skanują w tle powierzchnię dysku i zgłaszają informację o napotkanych uszkodzeniach.

- liczba realokacji (ang. *reallocation count*) — jeżeli napęd napotka błędny sektor to może zmienić jego logiczny numer pobierając sprawny sektor z puli zapasowej. Liczba realokacji mówi ile razy taka sytuacja miała miejsce.

- realokacje w tle (ang. *offline reallocation*) — jest to podzbiór liczby realokacji, zawierający tylko te realokacje, które zdarzyły się w czasie skanowania w tle, nie zaś w czasie wykonywania rzeczywistych operacji wejścia-wyjścia.
- liczba wątpliwych sektorów (ang. *probational count* — jeżeli jakiś sektor wykazuje oznaki nieprawidłowego działania to napęd umieszcza go na liście wątpliwych sektorów do czasu, aż ulegnie uszkodzeniu lub zacznie działać prawidłowo. Liczba wątpliwych sektorów może być traktowana jako wczesny system ostrzegania przed nadchodzącymi problemami.

Nawet gdyby zsumować wszystkie problemy sygnalizowane przez S.M.A.R.T., w dalszym ciągu 36% zepsutych dysków nie dawało żadnych oznak nadchodzącej awarii.



Rysunek 1.2: Procentowy udział błędów zgłoszonych przez S.M.A.R.T. w ogólnej liczbie uszkodzonych dysków [PWB07]

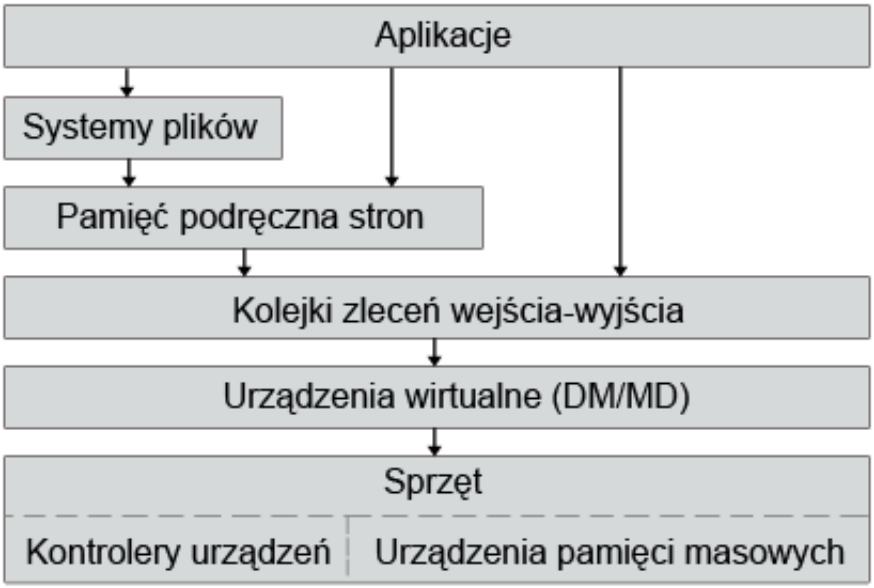
Jak widać w przytoczonych pracach awarie dysków, pomimo ciągłego postępu technologicznego, są cały czas poważnym problemem. Nawet jeżeli udałoby się stworzyć całkowicie bezawaryjny dysk, w dalszym ciągu nie możemy mieć pewności, że nie dojdzie do uszkodzenia danych. Na uszkodzenia nośników mają bowiem wpływ warunki zewnętrzne w jakich muszą one pracować.

Rozdział 2

Mechanizmy symulowania awarii

W trakcie tworzenia oprogramowania systemowego dużą wagę przywiązuje się do jego poprawności i wydajności. Końcowy użytkownik musi mieć pewność, że używany system działa zgodnie ze specyfikacją, a także wydajnie w standardowych zastosowaniach. W przypadku podsystemu obsługi urządzeń blokowych istnieje konieczność testowania wielu elementów systemu, począwszy od obsługi systemu plików, poprzez podsystem przekazywania zleceń wejścia-wyjścia, a na obsłudze urządzeń blokowych, takich jak dyski, kończąc. Istnieje wiele rozwiązań służących do badania zachowania podsystemu blokowego, część z nich znajduje się w jądrze systemu, część może być uruchamiana przez użytkownika. Niestety większość programów testujących służy do badania jak dana część podsystemu zachowuje się w normalnych warunkach pracy lub w przypadku systemów plików, czy spełnia standardy wymagane przez specyfikację. W kolejnych punktach przedstawię dostępne mechanizmy testowania różnych części systemu obsługujących operacje blokowe. Rozpocznę jednak od opisania budowy podsystemu obsługi urządzeń blokowych, na przykładzie systemu Linux.

2.1. Elementy podsystemu blokowego



Rysunek 2.1: Uproszczony schemat podsystemu blokowego

Za obsługę urządzeń pamięci masowych w systemie operacyjnym odpowiedzialny jest podsystem blokowy. W Linuksie, w jego skład wchodzą pamięć podręczna stron (ang. *Page Cache*), kolejki zleceń wejścia-wyjścia, sterowniki protokołów komunikacji z urządzeniami blokowymi, sterowniki kontrolerów urządzeń i sterowniki urządzeń (por. rys. 2.1). Odwołania do urządzeń blokowych mogą być generowane bezpośrednio przez aplikacje lub pośrednio przez system plików. Domyślnie wszystkie zlecenia przechodzą przez pamięć podręczną, dzięki czemu ponowny odczyt lub zapis może być znacznie szybszy w przypadku gdy dane nie uległy zmianie między kolejnymi operacjami. Wyjątkiem od tego jest tryb bezpośredniego dostępu do urządzeń, pomijający pamięć podręczną. Został on wprowadzony na potrzeby specjalizowanych aplikacji, takich jak bazy danych, dla których standardowy algorytm obsługi pamięci podręcznej może nie dawać wzrostu wydajności, a nawet ją pogarszać.

Wszystkie zlecenia zapisu/odczytu trafiają do kolejek zleceń obsługiwanych przez specjalny wątek jądra. Każde zlecenie opisywane jest przy pomocy struktury *bio*, która zawiera informacje o docelowym urządzeniu, typie operacji, położeniu i rozmiarze danych na urządzeniu, adresie w pamięci operacyjnej do/z którego mają być skopiowane dane.

Każde zlecenie *bio* może zostać zmodyfikowane przez sterowniki urządzeń logicznych MD (*Multiple Devices*) i DM (*Device Mapper*). W najprostszym przypadku modyfikacja zlecenia może polegać na zmianie docelowego urządzenia blokowego lub zastąpieniu go innym zleceniem. Przykładowo sterownik *dm-crypt* szyfruje dane w czasie zapisu i deszyfruje w czasie odczytu, sterownik *dm-raid1* rozkłada zlecenia zapisu na dwa dyski, realizując replikację danych.

Ostatnim etapem jest obsługa zlecenia przez sterowniki odpowiedzialne za obsługę sprzętu. Zlecenie jest tłumaczone na język protokołu właściwego dla danego urządzenia, a następnie przekazywane do sterownika kontrolera i, na koniec, do urządzenia blokowego. Po wykonaniu operacji sterowniki te zwracają informację o stanie, który jest przekazywany do zleceniodawcy.

2.2. Śledzenie pracy podsystemu blokowego

Większość nowoczesnych systemów operacyjnych posiada mechanizmy pozwalające na śledzenie wykonywanych operacji blokowych. W systemie Linux taką funkcjonalność zapewnia *blktrace*, czyli *Block I/O Layer Tracing*, umożliwiające śledzenie wszystkich operacji jakie są wykonywane na zleceniach wejścia-wyjścia. W skład *blktrace* wchodzi mechanizm wewnątrz jądra systemu emitujący zdarzenia informacyjne na każdym etapie obsługi zlecenia, a także programy działające w przestrzeni użytkownika służące do odczytywania tych informacji, ich prezentacji i konfiguracji mechanizmu systemowego. Przy pomocy *blktrace* można badać co dzieje się z każdym zleceniem odczytu lub zapisu od jego powstania, czy musiało zostać podzielone na mniejsze zlecenia, połączone z innymi (np. dla zwiększenia wydajności), aż do jego zakończenia. Nie da się jednak z jego pomocą uzyskać informacji o wpływie awarii na inne podsystemy zależne od warstwy blokowej.

Informacje uzyskane przy pomocy mechanizmu śledzącego można wykorzystać do modelowania pracy podsystemu blokowego. W pracy [SSFZ99] zaprezentowano porównanie rzeczywistego działania systemu i symulatora działającego w oparciu o ślad. Dzięki bardzo wiernemu odwzorowaniu pracy systemu przez symulator sterowany śladem można testować, między innymi, zmiany w obsłudze pamięci podręcznej lub nowe rozwiązania w systemach plików.

2.3. Symulatory dysków

Innym rozwiązaniem są symulatory dysków. Taki symulator jest programem działającym poza jądrem, który stara się emulować wszystkie aspekty działania dysku, takie jak liczba ścieżek, sektorów i głowic czy pamięć podręczną, a także współpracę między wszystkimi elementami dysku, aby emulowany dysk jak najwierniej symulował pracę rzeczywistego urządzenia, tak w warstwie sprzętowej, jak i logiki sterującej. Najbardziej rozbudowane rozwiązania pozwalają emulować nie tylko dysk, ale i całą infrastrukturę sprzętową odpowiedzialną za jego działanie.

Najlepszym przykładem jest program *DiskSim* stworzony na uniwersytecie Carnegie Mellon [DS40]. Umożliwia on pełną symulacje działania systemu obsługi dysków. Symulowane przez *DiskSim* środowisko składa się z następujących elementów:

- sterownik systemowy,
- szyna systemowa,
- kontroler dysków,
- dysk,
- kolejki wejścia-wyjścia,
- pamięć podręczna,
- urządzenia obsługujące pamięć podręczną.

Każdy z tych elementów posiada dużą liczbę parametrów konfiguracyjnych, pozwalających na tworzenie dowolnych środowisk testowych.

DiskSim wykorzystywany był w wielu opracowaniach dotyczących wydajności nowoczesnych systemów przechowywania danych [Worthington94] i ich wpływu na ogólną wydajność systemu [Ganger95], a także do oceny nowych architektur systemów przechowywania danych [Schlosser03].

Ponieważ *DiskSim* służy do modelowania zachowania systemów dyskowych i badania ich wydajności nie da się go wykorzystać do badania wpływu awarii na inne podsystemy, ponieważ nie przechowuje on danych, a tylko dostarcza informacje dotyczące funkcjonowania systemu.

2.4. Sterownik *faulty* podsystemu MD w Linuksie

Jądro systemu Linux posiada sterownik pozwalający na symulowanie pewnych rodzajów awarii urządzenia blokowego. Jest on częścią podsystemu MD.

2.4.1. Podsystem MD

Podsystem MD, czyli *Multiple Devices*, pozwala na tworzenie logicznych urządzeń blokowych zbudowanych z innych urządzeń blokowych — tak fizycznych, jak logicznych. Wykorzystując podsystem MD można tworzyć macierze RAID i konfigurować wielościeżkowy dostęp do dysków.

2.4.2. Sterownik *faulty*

W skład podsystemu MD wchodzi również sterownik *faulty*, który umożliwia generowanie zdarzeń błędu odczytu i/lub zapisu. Przy pomocy tego sterownika można symulować następujące błędy:

- zapis zawodzi dla losowych sektorów, powtórny zapis w to samo miejsce udaje się,
- odczyt zawodzi dla losowych sektorów, powtórny odczyt tego samego miejsca udaje się,
- odczyty nie powodzą się dopóty, dopóki nie nastąpi zapis,
- zapis zawodzi dla losowych sektorów,
- odczyt zawodzi dla losowych sektorów,
- wszystkie zapisy nie powodzą się.

Przy czym losowość oznacza co n-te zlecenie zapisu lub odczytu, w przypadku gdy n=0 zdarzenie błędu generowane jest tylko jeden raz. Można skonfigurować kilka trybów pracy jednocześnie, a także wyzerować konfigurację.

Sterownik *faulty* posiada kilka ograniczeń. Przede wszystkim generowane błędy nie powodują, w przypadku zapisu, że dane do zapisania nie trafią na dysk. Jedynie w trybie awarii dla wszystkich zapisów dane nie są zapisywane. Takie zachowanie sterownika powoduje, że możliwa jest tylko analiza zachowania wyższych warstw obsługi urządzeń blokowych, natomiast nie da się zbadać w jakim stopniu uszkodzeniu uległy dane na nośniku. Innym ograniczeniem *faulty* jest brak pełnej pamięci na temat wygenerowanych błędów, sterownik przechowuje informację na temat 50 zdarzeń, w przypadku gdy na nowe zdarzenie nie ma miejsca nie jest ono generowane. Nie można również spowodować, aby awarii ulegały nielosowe, ściśle określone obszary. Sterownik ten spełnia swoje zadanie jako narzędzie do testowania poprawności danych odtwarzanych przez algorytmy sum kontrolnych dla macierzy RAID [Szlufik06], a także odporności tych macierzy na losowe awarie, natomiast, z powodu swoich ograniczeń, niemożliwe jest wykorzystanie go do bardziej wymagających testów.

2.5. Rozwiązania specjalizowane

Ponieważ istniejące mechanizmy testowania nie spełniają wymagań koniecznych do szczegółowych badań wpływu awarii na systemy plików, powstają rozwiązania specjalizowane dla konkretnych typów testów. Przykładem są sterowniki napisane na potrzeby pracy [ViPr06], których zadaniem było:

- obserwacja zleceń zapisu i odczytu generowanych przez system plików,
- analiza do jakich struktur danych systemu plików należą zapisywane bloki,
- komunikacja z procesem generującym testy,
- generowanie błędów zapisu i odczytu.

Przy pomocy tych sterowników autor badał czy najpopularniejsze systemy plików (NTFS dla Windows, ext3, ReiserFS w wersji 3, JFS i XFS dla Linuksa) poprawnie wykrywają i obsługują błędy wejścia-wyjścia. W tym celu każdy sterownik posiadał dokładną informację na temat budowy testowanego systemu plików. Wyniki testów pokazały, że różne systemy plików zachowują się zupełnie inaczej w przypadku błędów, a w skrajnych przypadkach zdarza

się, że informacja o błędzie jest gubiona. Najlepiej wypadły NTFS, XFS i ReiserFS, każdy z nich dobrze wykrywa i zgłasza błędy, przy czym NTFS najpierw stara się powtórzyć kilkukrotnie operację. XFS przy błędach zapisów synchronicznych zatrzymuje system, a dla asynchronicznych powtarza operację do skutku, natomiast ReiserFS w każdym przypadku zatrzymuje system aby zapewnić spójność danych. Gorzej wypadły ext3 i JFS, które w pewnych sytuacjach ignorowały błędy zapisu lub, w przypadku ext3, próbowały zatwierdzać błędne transakcje.

Z powodu bardzo dużego stopnia specjalizacji tych sterowników nie można ich jednak wykorzystać do jakichkolwiek innych testów. Także może się zdarzyć, że próba użycia ich z innym jądrem niż to, dla którego zostały napisane nie powiedzie się i spowoduje przekłamania wyników, ponieważ zmianie mogły ulec struktury wewnętrzne systemu plików.

Rozdział 3

Projekt i implementacja sterownika

3.1. Założenia

Ponieważ żadne z dostępnych rozwiązań nie umożliwia w pełni realizacji założonego celu, którym jest symulowanie różnego rodzaju awarii w kontrolowany sposób, powstała potrzeba stworzenia nowego sterownika.

Głównym założeniem przy projektowaniu sterownika była chęć wykonania testów odporności na awarie dla systemów plików. Jak istotny jest to problem widać we wspomnianej wcześniej pracy [ViPr06]. Sterowniki obsługujące systemy plików są jedną z kluczowych części systemu operacyjnego co powoduje, że chcemy wiedzieć jak zachowają się w sytuacji ekstremalnej, jaką jest uszkodzenie nośnika. Testy takie mogą również pomóc zlokalizować i naprawić błędy, które nie zostały wykryte w czasie analizy kodu oraz standardowych prób poprawności i wydajności. Jednocześnie sterownik powinno dać się wykorzystać do testowania funkcjonalności innych sterowników urządzeń logicznych, a także programów, które używają urządzeń blokowych z pominięciem systemu plików (np. bazy danych).

W związku z tym implementacja sterownika musiała spełniać następujące wymagania:

- niezależność od sprzętu — możliwość przeprowadzenia testów nie powinna być zależna od posiadanego sprzętu, a także zachowanie sprzętu nie powinno wpływać na przeprowadzane testy i ich wyniki.
- konfigurowalność — umożliwienie pełnej kontroli nad zachowaniem urządzenia i tworzenie rozbudowanych scenariuszy testowania.
- łatwość użycia — nawet najlepszy program nie będzie wykorzystywany, jeśli jego obsługa będzie zbyt skomplikowana lub będzie wymagał specjalistycznych narzędzi.

3.1.1. Podsystem blokowy w Linuksie

W trakcie projektowania sterownika do symulacji awarii musiałem rozważyć, w której warstwie podsystemu blokowego umieścić sterownik. Do wyboru były następujące rozwiązania:

1. Modyfikacja istniejącego sterownika obsługującego dyski.
2. Napisanie własnego sterownika na podstawie urządzenia *loop*.
3. Rozbudowanie sterownika *faulty* podsystemu MD o wymaganą funkcjonalność.
4. Napisanie sterownika mapującego wykorzystującego podsystem DM.

Rozwiązanie pierwsze zapewnia praktycznie całkowite wyeliminowanie wpływu systemu operacyjnego na wyniki testów, ponieważ generowanie błędów odbywa się na najniższym możliwym poziomie. Powoduje jednak uzależnienie od konkretnego rodzaju dysków, co ogranicza możliwości jego stosowania, jest również rozwiązaniem inwazyjnym, bardzo głęboko ingerującym w kod jądra, co utrudnia jago wykorzystanie. W drugim rozwiązaniu oprócz obsługi urządzenia blokowego, należy również wziąć pod uwagę interakcję z systemem plików, na którym znajduje się plik udostępniany jako urządzenie, co powoduje dodatkową komplikację. Rozwiązanie to ogranicza też możliwości wykorzystania sterownika. Z dwóch ostatnich rozwiązań mój wybór padł na podsystem *device mapper*, gdyż próba rozbudowania sterownika *faulty* wymagałaby w gruncie rzeczy przepisania go od nowa, jak również wprowadzenia dużych modyfikacji w programie administracyjnym dla urządzeń MD. W przypadku podsystemu *device mapper*, dzięki dobrze zdefiniowanemu protokołowi komunikacyjnemu, nie ma potrzeby modyfikacji narzędzi systemowych. Wydał mi się on też bardziej naturalnym miejscem dla urządzenia, którego zadaniem jest tylko modyfikacja zleceń.

3.1.2. *Device Mapper*

Device mapper (w skrócie DM) jest podsystemem umożliwiającym tworzenie logicznych urządzeń blokowych, przy pomocy których można przekierowywać zlecenia zapis/odczyt na inne urządzenie. Za obsługę urządzeń tego typu odpowiada sterownik jądra *dm-mod*, który na podstawie informacji o urządzeniu przekazuje sterowanie do odpowiednich modułów. Każdy moduł musi dostarczać przynajmniej podstawowe funkcje zdefiniowane w interfejsie podsystemu *device mapper*. Przy pomocy podsystemu DM można tworzyć stos urządzeń mapujących, łącząc ich funkcjonalność bez potrzeby tworzenia nowych sterowników. Obecnie, w jądrze Linuksa, dostępne są, między innymi, moduły umożliwiające łączenie wielu dysków (*dm-liear*), szyfrowanie (*dm-crypt*) czy wielościeżkowy dostęp do dysków (*dm-multipath*). Kod tych i pozostałych modułów, a także modułu sterującego *dm-mod* znajduje się w katalogu *drivers/md*, w źródłach jądra systemu Linux.

3.1.3. Interfejs podsystemu *Device Mapper*

Urządzenia

Każde urządzenie podsystemu DM jest reprezentowane przez strukturę *dm_target*:

```
struct dm_target {
        struct dm_table *table;
        struct target_type *type;
        sector_t begin;
        sector_t len;
        sector_t split_io;
        struct io_restrictions limits;
        void *private;
        char *error;
};
```

- `table` — wskaźnik do tablicy konfiguracyjnej zawierającej to urządzenie,
- `type` — wskaźnik do interfejsu urządzenia danego typu,
- `begin` — początek danych na urządzeniu mapującym,

- `len` — wielkość urządzenia wyrażona w 512 bajtowych blokach,
- `split_io` — liczba bloków w pojedynczym zleceniu,
- `limits` — ograniczenia specyficzne dla mapowanego urządzenia,
- `private` — wskaźnik do prywatnych danych sterownika,
- `error` — miejsce na komunikat błędu dla funkcji tworzącej mapowanie.

Sterowniki

Sterownik urządzenia DM opisuje struktura `target_type`:

```
struct target_type {
        const char *name;
        struct module *module;
        unsigned version[3];
        dm_ctr_fn ctr;
        dm_dtr_fn dtr;
        dm_map_fn map;
        dm_endio_fn end_io;
        dm_flush_fn flush;
        dm_presuspend_fn presuspend;
        dm_postsuspend_fn postsuspend;
        dm_preresume_fn preresume;
        dm_resume_fn resume;
        dm_status_fn status;
        dm_message_fn message;
        dm_ioctl_fn ioctl;
};
```

Pole `name` zawiera nazwę mapowania, `version` wersję modułu, `module` wewnętrzne informacje jądra na temat modułu. Pozostałe pola struktury to wskaźniki do funkcji implementujących mapowanie.

Najważniejszymi funkcjami interfejsu DM są:

```
int (*dm_ctr_fn) (struct dm_target *target,
                  unsigned int argc, char **argv);
```

Funkcja tworząca urządzenie mapujące na podstawie dostarczonych parametrów. Odpowiada za konfigurację pseudourządzenia, alokację niezbędnej pamięci i zainicjowanie struktur danych. Struktura `dm_target` w czasie wywołania ma ustawione najważniejsze pola — `table`, `type`, `begin` i `len`.

```
void (*dm_dtr_fn) (struct dm_target *ti);
```

Funkcja usuwająca mapowanie. Nie zwalnia struktury `dm_target`, natomiast musi posprzątać wszelkie dane, które znajdują się w `ti→private`.

```
int (*dm_map_fn) (struct dm_target *ti, struct bio *bio,
                  union map_info *map_context);
```

Podstawowa funkcja odpowiedzialna za obsługę zleceń zapisu i odczytu. W zależności od podjętego działania przekazuje:

- < 0 — błąd,
- 0 — urządzenie samodzielnie obsłuży zlecenie,
- 1 — proste mapowanie zakończone,
- 2 — zlecenie ma zostać cofnięte do kolejki.

Interfejs DM umożliwia komunikację między programami użytkownika a sterownikami przy pomocy następujących funkcji:

```
int (*dm_message_fn) (struct dm_target *ti, unsigned argc, char **argv);
```

Funkcja obsługująca interfejs komunikatów DM. Pozwala na odbieranie prostych komunikatów tekstowych od programu konfiguracyjnego *dmsetup*.

```
int (*dm_ioctl_fn) (struct dm_target *ti, struct inode *inode,
                    struct file *filp, unsigned int cmd,
                    unsigned long arg);
```

Obsługa standardowego interfejsu I/O Control, umożliwiającego programom komunikację i sterowanie urządzeniem.

Pozostałe funkcje interfejsu DM:

- `dm_status_fn` jest wykorzystywana do przedstawienia ogólnych informacji na temat urządzenia.
- `dm_presuspend_fn`, `dm_postsuspend_fn`, `dm_preresume_fn`, `dm_resume_fn` służą do wprowadzania i wyprowadzania urządzenia ze stanu uśpienia.
- `dm_endio_fn` obsługuje zdarzenie zakończenia obsługi zlecenia.

Wejście-wyjście

Ponieważ w urządzeniach podsystemu DM występuje konieczność wykonywania operacji na blokach danych, stworzony została interfejs *dm-io* udostępniający funkcje do synchronicznego i asynchronicznego zapisu i odczytu sektorów. Pozwala on w wygodny sposób manipulować blokami dyskowymi bez konieczności pisania obsługi zleceń *bio*. Dodatkowo umożliwia on zapisywanie tych samych danych na więcej niż jedno urządzenie bez konieczności tworzenia wielu zleceń.

Zlecenia dm-io realizowane są za pomocą funkcji:

```
int dm_io(struct dm_io_request *io_req, unsigned num_regions,
          struct dm_io_region *where, unsigned long *sync_error_bits)
```

Parametrami tej funkcji są:

- `dm_io_request` — zawierają kierunek operacji, adres do obszaru pamięci, a także wskaźnik do funkcji, która zostanie wykonana po zakończeniu operacji w trybie asynchronicznym.
- `num_regions` — określa ilu obszarów dotyczy zlecenie.
- `dm_io_region` — lista obszarów, na których wykonywana jest operacji. Zawiera urządzenie, początkowy sektor i liczbę sektorów.
- `sync_error_bits` — maska bitowa określająca, dla których obszarów wystąpił błąd w przypadku zleceń synchronicznych.

3.2. Sterownik dm-fault

3.2.1. Implementacja

Pierwszą rzeczą jaką musiałem rozważyć podczas pisania sterownika było przechowywanie informacji o stanie sektorów. Najprostszą metodą byłaby tablica uszkodzonych sektorów w pamięci operacyjnej. Założyłem jednak, że nie będę ograniczał liczby równocześnie symulowanych uszkodzeń, a tablica ta mogłaby nie zmieścić się w pamięci. Z tego powodu uznałem, że najlepszym wyjściem będzie przechowywanie danych na dysku, dzięki czemu można symulować dowolną liczbę uszkodzeń jednocześnie. Pewnymi wadami tego rozwiązania są spadek wydajności, spowodowany koniecznością wykonania dodatkowej operacji zapisu i odczytu dla każdego zlecenia, oraz zmniejszenie wielkości udostępnionego dysku. Przy obecnych rozmiarach dysków, jak również w warunkach testowych, utrata około 12% pojemności nie powinna być problemem. Zmniejszenie wydajności też nie jest tak istotne jak mogłoby się wydawać, gdyż przy symulowaniu awarii ważniejsza jest wiarygodność testu niż szybkość jego wykonania. Do obsługi składowania stanu na dysku wykorzystałem interfejs *dm-io*, pozwalający na wykonywanie synchronicznych operacji bez użycia interfejsu *bio*. Aby jednak umożliwić przeprowadzanie testów w warunkach wymagających wydajnej pracy dysków, udostępniłem mechanizm do wyłączania zachowywania stanu. Powoduje to niestety niedostępność tych trybów pracy, dla których informacja ta jest niezbędna.

Wybór dysku jako miejsca przechowywania danych spowodował, że niemożliwym stało się wykorzystanie prostego mapowania w funkcji `dm_map`. Każdy proces posiada własny stos w przestrzeni jądra, który w przeciwieństwie do zwykłego stosu nie jest powiększany w razie potrzeby. Ponieważ ciąg wywołań funkcji obsługujących zlecenia zapisu i odczytu może być bardzo długi w przypadku wykorzystania urządzeń logicznych, należy oszczędnie gospodarować pamięcią dostępną na stosie. Wykonanie dodatkowych operacji w czasie obsługi zlecenia mogłoby spowodować przepełnienie stosu jądra dla procesu, który zlecenie wygenerował. Oprócz tego rekurencyjne wykonywanie zleceń wejścia-wyjścia i manipulacja zawartością sektorów po ich odczytaniu w funkcji obsługującej zakończenie obsługi zlecenia nie są zalecanymi praktykami. W związku z tym uznałem, że bezpiecznym rozwiązaniem będzie utworzenie specjalnego wątku jądra, który będzie odpowiedzialny za obsługę zleceń. Wszystkie żądania przychodzące są odkładane do kolejki zleceń, następnie osobny wątek zajmuje się ich obsługą.

W czasie pracy sterownik wymaga pewnej ilości pamięci na swoje struktury danych i na informacje o stanie wczytywane z dysku. W jądrze Linuksa istnieje kilka interfejsów dynamicznej alokacji pamięci. Linux pozwala na alokację pojedynczych stron (`page_alloc()`), ciągłego obszaru pamięci fizycznej (`kmalloc()`) lub wirtualnej (`vmalloc`). Interfejsy te mają tę wadę, że w przypadku większego obciążenia podsystemu obsługi pamięci funkcje alokujące mogą zostać zablokowane w oczekiwaniu na wolną pamięć. Dlatego zdecydowałem się skorzystać z interfejsu mempool, który umożliwia alokację bez oczekiwania poprzez rezerwowanie puli pamięci. W przypadku gdy niemożliwa jest alokacja bez oczekiwania, pamięć pobierana jest z przygotowanej puli. W moim sterowniku wykorzystuję predefiniowane funkcje do alokacji stron, niezbędnych do wczytywania danych z dysku, a także do tworzenia obiektów w pamięci podręcznej jądra w celu przechowywania dynamicznych struktur danych sterownika.

Ponieważ funkcjonowanie sterownika jest kontrolowane przez użytkownika, musiałem zdecydować czy skorzystać z interfejsu komunikatów tekstowych podsystemu *device mapper*, czy ze standardowego ioctl (ang. *I/O control*). Początkowo zamierzałem używać ioctl co umożliwiłoby przekazywanie sterownikowi gotowej struktury konfiguracyjnej. Wadą tej metody jest konieczność używania specjalnego programu do kontrolowania urządzenia, dlatego zdecydowałem jednak wykorzystać interfejs komunikatów, gdyż pozwala on używać ogólnie dostęp-

nego programu *dmsetup* a jednocześnie nie wyklucza możliwości stworzenia specjalizowanego oprogramowania.

Sterownik został napisany dla Linuksa w wersji 2.6.26 i z tą wersją był testowany. Jądro systemu Linux podlega ciągłemu rozwojowi, zmieniają się interfejsy programistyczne, położenie i nazwy plików nagłówkowych. Z tego powodu może być niemożliwe skompilowanie lub uruchomienie sterownika z inną wersją Linuksa.

3.2.2. Struktury danych

Podsystem DM umożliwia tworzenie nieograniczonej liczby urządzeń, w związku z tym sterownik musi dla każdego urządzenia przechowywać informacje o stanie i konfiguracji. Do tego celu służy struktura `fault_c`:

```
struct fault_c {
        struct dm_dev *dev;
        sector_t info_start;
        sector_t info_size;
        int state;
        struct dm_io_client *io_client;
        mempool_t *pl_pool;
        mempool_t *page_pool;
        mempool_t *io_pool;
        struct workqueue_struct *kfaultd_wq;
        struct work_struct kfaultd_work;
        spinlock_t lock;
        struct bio_list requests;
        struct bio_list reads;
        struct mutex sem;
        int config_mode;
        struct fault_conf_list config;
        sector_t lbio_sector;
        unsigned short lbio_vcnt;
        unsigned short lbio_idx;
        unsigned int lbio_size;
};
```

Pole `dev` wskazuje na strukturę zawierającą informacje na temat mapowanego dysku, takie jak jego nazwa i wskaźnik do oryginalnego urządzenia blokowego. Wartością `info_start` jest numer sektora, od którego zaczynają się dane o stanie sektorów, a `info_size` to liczba sektorów przeznaczona na przechowywanie stanu. Pole `state` informuje czy sterownik ma zachowywać stan. Aby podsystem *dm-io* mógł poprawnie działać niezbędne jest zainicjowanie pewnych danych, przechowywane są one w strukturze `io_client`. Pola `pl_pool`, `page_pool` i `io_pool` zawierają wskaźniki do obszarów pamięci zarezerwowanych dla obsługi zapisów i odczytów wykonywanych przez sterownik. Każde z urządzeń posiada własny wątek obsługujący zlecenia, stan wątku zawarty jest w strukturach `kfaultd_wq` i `kfaultd_work`. Wszystkie przychodzące zlecenia odkładane są na liście `requests`. Lista `reads` zawiera zlecenia odczytu, które muszą być dodatkowo przetworzone po wykonaniu operacji. Aby zapobiec wyścigowi w momencie gdy zlecenia są obsługiwane przez wątek sterownika, listy muszą być chronione przed zapisem przy pomocy blokady `lock`. Ponieważ założyłem że konfiguracja urządzenia

ma być elastyczna, pojedyncza struktura z parametrami konfiguracji byłaby niewystarczająca, dlatego przechowywane są ona na liście, do której dostęp odbywa się przez pole `config`. Semafor `sem` zabezpiecza listę konfiguracyjną przed modyfikacją w trakcie zmiany konfiguracji lub obsługi zlecenia. Parametr `config_mode` informuje sterownik jak traktować przypadki gdy dany sektor należy do kilku różnych konfiguracji. W `lbio_sector`, `lbio_vcnt`, `lbio_idx` i `lbio_size` przechowywana jest kopia najważniejszych informacji dotyczących ostatniego wykonanego zlecenia zapisu.

Dane o stanie sektorów przechowywane są na dysku w postaci tablicy rekordów typu `struct fault_status`, indeksowanej numerem sektora. Wybór dysku jako miejsca przechowywania informacji o stanie spowodował, że szczególną uwagę musiałem poświęcić strukturze opisującej te dane. Ponieważ dane z dysku są wczytywane i zapisywane sektorami należało zadbać, aby rozmiar sektora, o standardowej dla Linuksa wielkości 512 bajtów, był podzielny bez reszty przez rozmiar struktury. W innym wypadku dostęp do danych byłby skomplikowany z powodu konieczności obsługi warunków brzegowych gdy struktura znalazłaby się na granicy sektora lub strony w pamięci. Stąd też konieczność dodania „wypełniaczy" zapewniających rozmiar 64 bajtów.

```
struct fault_status {
        u32 rretry;
        u32 wretry;
        u64 owners;
        u64 perm_fix;
        u64 reserved;
};
```

W elementach `rretry` i `wretry`, zapisywana jest liczba ponowień prób odczytu i zapisu. Pola `owners` i `perm_fix` są maskami bitowymi, których bity odpowiadają numerom identyfikacyjnym konfiguracji, mówią one która konfiguracja zawiera dany sektor i dla której jest on permanentnie naprawiony. Użycie mapy bitowej oznacza że liczba jednoczesnych konfiguracji nie może przekroczyć 64. Uważam jednak że jest to opłacalny kompromis. Po pierwsze nieograniczona liczba konfiguracji mogłaby spowodować że niemożliwym stałoby się utrzymywanie informacji o stanie każdego sektora dla każdej konfiguracji. Po drugie istnieje możliwość nakładania na siebie urządzeń dm-fault, co w efekcie daje pełną swobodę działania.

Opisy każdej konfiguracji przechowywane są w strukturach `fault_conf`.

```
struct fault_conf {
        int id;
        uint32_t mode;
        int32_t period;
        uint64_t start;
        uint64_t size;
        uint64_t step;
        uint32_t nr_steps;
        uint32_t retries;
        uint32_t noise;
        uint32_t io_count;
        struct fault_conf_k *c_next;
};
```

Każda konfiguracja posiada swój unikatowy identyfikator (pole `id`) z zakresu [0..63]. Obszar, którego dotyczy dana konfiguracja, opisywany jest przez pola `start`, `size`, `step` i `nr_steps`, natomiast pole `mode` opisuje zachowanie dla danego obszaru. Elementy `period` i `io_count` wykorzystywane są w przypadku gdy uszkodzenie ma dotyczyć co n-tego lub losowego zlecenia, gdzie `period` oznacza n, jeśli jest liczbą dodatnią, lub prawdopodobieństwo w procentach, jeśli jest liczbą ujemną, a `io_count` jest licznikiem zleceń. Element `retries` oznacza liczbę powtórzeń po ilu zlecenie ma się powieźć, o ile wybrany został ten tryb pracy. Podobnie `noise` jest liczbą bajtów, które mają być zmienione w trybie zaszumiania danych. Wskaźnik `c_next` jest elementem pomocniczym do tworzenia listy konfiguracji.

3.2.3. Tryby pracy sterownika

Podstawową funkcjonalnością udostępnianą przez sterownik jest możliwość zablokowania zapisów lub odczytów na określonym obszarze dysku. Obszar definiowany jest przez początkowy sektor i rozmiar jakiego dotyczy. Dodatkowo można określić, że obszar ma być powtarzany co pewną liczbę sektorów, oraz że na danym obszarze błędy mają być generowane losowo z danym prawdopodobieństwem albo co określoną liczbę operacji. Blokada zapisu lub odczytu jest wykonywana przez przekazanie błędu wejścia-wyjścia dla operacji, która należy do obszaru skonfigurowanego jako uszkodzony. Sterownik umożliwia jednoczesne ustawienie 64 definicji obszarów, z których każdy może symulować inny rodzaj awarii. Obszary mogą się dowolnie przecinać, co w pewnych przypadkach może prowadzić do tego, że działanie sterownika będzie bardzo trudne do przewidzenia. Zdecydowałem, że nie będę zabezpieczał sterownika na wypadek takich sytuacji, gdyż byłoby to bardzo kosztowne obliczeniowo i wymagałoby zgadywania jaki efekt użytkownik chciał osiągnąć.

Do realizacji pełnej funkcjonalności sterownik przechowuje stan operacji w ukrytym obszarze dysku. W czasie tworzenia urządzenia blokowego istnieje możliwość wyłączenia zachowywania stanu, ale wtedy część trybów pracy i typów symulowanych awarii będzie niedostępna, lub ich działanie będzie inne od domyślnego.

Oprócz prostej blokady operacji sterownik umożliwia symulowanie pewnych typowych przypadków awarii:

- powtórzenia — operacja zapisu/odczytu dla danego sektora kończy się sukcesem dopiero przy N-tej próbie; ten typ awarii jest niedostępny przy wyłączonym zachowywaniu stanu.
- zapis naprawia błędy odczytu — wszystkie odczyty dla danego sektora kończą się niepowodzeniem dopóki nie nastąpi w tym miejscu zapis; ten typ awarii jest niedostępny przy wyłączonym zachowywaniu stanu.
- odczyt naprawia błędy zapisu — wszystkie zapisy dla danego sektora kończą się niepowodzeniem dopóki nie nastąpi w tym miejscu odczyt; ten typ awarii jest niedostępny przy wyłączonym zachowywaniu stanu.
- zaszumienie danych — w czasie odczytu lub zapisu ustalona liczba bajtów w sektorze jest zmieniana na losowe wartości.

Sterownik posiada kilka trybów pracy, definiujących sposób obsługi sektorów należących do ustalonego obszaru. Domyślnie wszystkie błędy są permanentne, co oznacza, że akcja określona przez typ awarii będzie wykonywana zawsze dla każdego sektora z obszaru. Jednak, w przypadku gdy wyłączone jest zachowywanie stanu, sektory wybrane losowo tylko raz

zwrócą błąd gdyż nie ma wtedy możliwości zapamiętania że dany sektor został wylosowany. W kolejnych punktach przedstawię pozostałe tryby pracy udostępniane przez sterownik.

- Tryb jednokrotny — akcja dla sektora wykonywana jest jednokrotnie, gdzie akcją może być dowolny z typów awarii opisanych wcześniej.

- Tryb wprowadzanie błędów — pozwala na określanie, dla których sektorów z tego obszaru ma być wykonywana akcja zdefiniowana przez symulowany typ awarii. Obszar skonfigurowany w tym trybie pracy domyślnie nie wykonuje zdefiniowanej akcji, aby akcja została wykonana dla sektora, sterownik musi otrzymać komunikat konfiguracyjny mówiący, który sektor ma „uszkodzić" lub „naprawić". W przypadku gdy tryb wprowadzania błędów zostanie skonfigurowany dla danego obszaru, lub jego części, wielokrotnie, pod uwagę będzie brana tylko pierwsza pasująca konfiguracja.

- Uszkodzenie ostatniego zapisu — nie jest to tryb pracy jako taki, a informacja dla sterownika, że w momencie dodawania konfiguracji ma wyzerować zawartość sektorów należących do ostatniej wykonanej operacji zapisu, o ile należą one do dodawanego obszaru.

Tryby wprowadzania błędów i jednokrotnej akcji są niedostępne, jeżeli wyłączone jest zachowywanie stanu dla urządzenia.

Ponieważ sterownik umożliwia definiowanie obszarów, które mogą posiadać części wspólne, zachodzi potrzeba określenia jak traktować sektory należące do wielu obszarów jednocześnie. Sterownik udostępnia następujące strategie postępowania:

- pierwsza pasująca — domyślna strategia, wykonywana jest akcja zdefiniowana przez pierwszą konfigurację, do której należy sprawdzany sektor.

- ostatnia pasująca — wykonywana jest akcja zdefiniowana przez ostatnia konfigurację, do której należy sprawdzany sektor.

- dowolna — dla sektora zgłaszany jest błąd, jeśli uzna go za błędny choć jedna konfiguracja, do której sektor należy.

- wszystkie — dla sektora zgłaszany jest błąd, jeśli jest uznany za błędny przez wszystkie konfiguracje, do których należy.

Opis konfiguracji urządzenia `dm-fault` i komunikatów konfiguracyjnych znajduje się na płycie dołączonej do pracy.

3.3. Biblioteka libdmfault

Do zarządzania sterownikiem wykorzystywany jest mechanizm komunikatów tekstowych podsystemu *Device Mapper*. Jest on bardzo wygodny w użyciu z wiersza komend systemu, ale w programach wymaga korzystania ze złożonego interfejsu biblioteki narzędziowej *libdevmapper*.

Z tego powodu napisałem prostą bibliotekę, opakowującą tworzenie komunikatów i przekazywanie ich do sterownika przy pomocy funkcji biblioteki *libdevmapper*. Funkcje udostępniane przez bibliotekę *libdmfault* pozwalają w prosty sposób konfigurować i kontrolować urządzenia `dm-fault`, co może znacznie ułatwić tworzenie specjalistycznych testów.

Biblioteka udostępnia następujące funkcje:

`int dmfault_setconf(char *device, int mode)`

Ustawia strategię zachowania `mode` dla pokrywających się obszarów urządzenia `device`.

`int dmfault_add(char *device, int mode, u_int64_t start, u_int64_t size, u_int64_t step, int nr_steps, int period, int retries, int noise)`

Dodaje konfigurację nowego obszaru dla urządzenia `device`, typ awarii i tryb pracy opisują parametry `mode`, `period`, `retries` i `noise`. Obszar jest zdefiniowany parametrami `start`, `size` `step` i `nr_steps`.

`int dmfault_clear(char *device)`

Usuwa wszystkie skonfigurowane obszary dla urządzenia `device`.

`int dmfault_poweroff(char *device, int destroy)`

Symuluje zdarzenie odcięcia zasilania przez zablokowanie zapisów i odczytów na całym urządzeniu `device`, parametr `destroy` informuje czy sterownik ma uszkodzić sektory należące do ostatniego zlecenia zapisu.

`int dmfault_fail(char *device, u_int64_t sector)`

Funkcja dla trybu wprowadzania błędów, oznacza sektor `sector` jako uszkodzony na urządzeniu `device`.

`int dmfault_fix(char *device, u_int64_t sector)`

Funkcja dla trybu wprowadzania błędów, oznacza sektor `sector` jako poprawny na urządzeniu `device`.

Rozdział 4

Testy

Istotnym elementem tworzenia każdego oprogramowania jest testowanie go w warunkach, w jakich będzie wykorzystywane. W przypadku sterowników systemowych jest to niezwykle ważne nie tylko w celu sprawdzenia czy sam sterownik działa zgodnie ze specyfikacją, a także czy poprawnie współpracuje z innymi elementami systemu operacyjnego. Testy wykonywałem tworząc urządzenie blokowe przy pomocy sterownika *loop*, który udostępnia plik jako dysk. Na tak uzyskanym dysku zakładałem mapowanie własnym sterownikiem dm-fault, a w części testów dodatkowo dodawałem mapowanie opóźniające wykonanie wszystkich operacji blokowych o zadany czas, przy użyciu sterownika *dm-delay*.

Ze względu na, możliwie szkodliwy, wpływ sterownika na system, testy wykonywałem przy pomocy narzędzi emulujących w pełni sprzęt komputerowy. Do testów wykorzystywałem programy VirtualBox i VMWare Workstation.

VirtualBox jest programem służącym do wirtualizacji sprzętu z rodziny Intel x86, udostępniamym przez firmę Sun Microsystems na licencji Open Source. Umożliwia on tworzenie wirtualnych maszyn, na których można uruchomić w pełni funkcjonalny system operacyjny.

VMWare Workstation jest oprogramowaniem komercyjnym tworzonym przez firmę VMWare. Również służy do wirtualizacji sprzętu z rodziny Intel x86. Do przeprowadzenia testów wystarcza VirtualBox. Programu VMWare Workstation używałem, ponieważ pewien scenariusz testowy powodował błąd jądra, którego nie byłem w stanie przypisać własnemu sterownikowi. Jak się okazało, dzięki zastosowaniu alternatywnego oprogramowania, VirtualBox w aktualnej wersji 1.6.4, posiada błąd w obsłudze tak zwanych „dzielonych folderów". „Dzielone foldery" to fragmenty systemu plików komputera, na którym uruchamiany jest emulator, udostępniane systemowi wewnątrz emulatora, przy pomocy specjalnego sterownika. Zaobserwowany przeze mnie błąd objawia się w przypadku utworzenia urządzenia *loop* na „dzielonym folderze", a następnie wykonania jakiejkolwiek operacji zapisu na nim. Zaobserwowane objawy sugerują, że sterownik odpowiedzialny za obsługę „dzielonych folderów" nie jest odporny na bezpośrednie manipulacje, pomijające lokalny system plików, na udostępnianym systemie plików.

4.1. Testy poprawności

Testy poprawności wykonywałem tworząc możliwie jak największą liczbę kombinacji trybów pracy sterownika i wykonując na utworzonym urządzeniu proste operacje zapisu i odczytu przy pomocy programu *dd*, którym można zapisać i odczytać dowolny blok na dysku.

Schemat testu:

1. Utworzenie pliku o wielkości 64 MB.

2. Konfiguracja urządzenia *loop* na utworzonym pliku.

3. Konfiguracja mapowania *dm-fault* na urządzeniu *loop*.

4. Ustawienie parametrów mapowania (obszary i typy awarii).

5. Wykonanie operacji odczytu i zapisu na zamapowanym urządzeniu, zarówno w miejscach uszkodzonych, jak i poprawnych.

6. Wyzerowanie parametrów mapowania.

7. Powtórzenie operacji od punktu 4 do 6 dla kolejnych trybów pracy i typów awarii.

8. Usunięcie mapowania, urządzenia *loop* i pliku.

Dzięki wykorzystaniu urządzenia *loop*, nie ma potrzeby rezerwowania dodatkowych zasobów sprzętowych, co znacznie ułatwia i przyśpiesza wykonanie testu. Punkty 4 i 5 są głównym elementem testu, gdyż pozwalają stwierdzić, czy sterownik zachowuje się zgodnie ze specyfikacją. Punkt 8 daje pewność, że sterownik nie pozostawia po sobie żadnych informacji, które mogłyby zakłócać jego pracę w bardziej skomplikowanych scenariuszach.

4.2. Testy transakcyjności systemów plików

Interesującym zagadnieniem jest zachowanie systemu plików w przypadku awarii, a zwłaszcza jaki wpływ na system plików wywołuje odcięcie zasilania. Istnieje bardzo niewiele odpowiedniej dokumentacji i testów dotyczących tego zagadnienia, co powoduje, że wszystko co wiemy o zachowaniu systemów plików w takich sytuacjach to głównie teorie i spekulacje. Jedynym znanym mi opracowaniem na temat wpływu awarii dysków na system plików jest praca [ViPr06], w której autor badał jaki wpływ na system plików ma uszkodzenie ściśle określonych bloków, np. jak system plików zareaguje w przypadku niemożności odczytania lub zapisania metadanych w czasie wykonywania standardowych operacji.

W swojej pracy postanowiłem wykonać testy transakcyjności systemów plików, mające na celu sprawdzenie, jaki wpływ na spójność metadanych ma zdarzenie odcięcia zasilania, a także czy system plików zachowuje się poprawnie w przypadku zamontowania go w trybie synchronicznego zapisu. Do testów wybrałem najpopularniejsze systemy plików w Linuksie — ext3, XFS, ReiserFS. Wszystkie one są systemami plików z dziennikowaniem, które powinno zapewnić spójność metadanych w przypadku awarii.

Schemat testu:

1. Utworzenie pliku o wielkości 128 MB.

2. Konfiguracja urządzenia *loop* na utworzonym pliku.

3. Konfiguracja mapowania *dm-fault* na urządzeniu *loop*.

4. Założenie systemu plików na zamapowanym urządzeniu.

5. Zamontowanie systemu plików w trybie domyślnym lub *sync* i/lub *dirsync*.

6. Wykonanie operacji (create, rename, delete, append etc.).

7. Zablokowanie zapisu na urządzenie bezpośrednio po zakończeniu sekwencji wywołań systemowych.

(a) Opcjonalne — Wyzerowanie bloków w ostatnim wykonanym zleceniu zapisu.

8. Sprawdzenie czy operacja wykonana przed awarią jest widoczna.

Podobnie jak przy testach poprawności, tak i tu, aby ułatwić i przyśpieszyć wykonanie testu używam urządzenia *loop*. Montowanie systemu plików w trybie *sync* powoduje, że wszystkie zlecenia wejścia-wyjścia wykonywane są synchronicznie. Natomiast w trybie *dirsync*, synchronicznie wykonywane są tylko operacje dotyczące katalogów, takie jak utworzenie pliku, katalogu lub zmiana nazwy. Montowanie systemu plików w trybie synchronicznego zapisu powinno gwarantować, że wszystkie operacje wykonane przed zablokowaniem dostępu do urządzenia znajdą się na dysku. Dodatkowo część operacji jest wykonywana na plikach otwartych w trybie synchronicznym (przez użycie flagi `O_SYNC` w funkcji `open()`) i/lub jest potwierdzana przy pomocy funkcji `fsync()`, która wymusza zapis na dysk. Jeżeli awaria zasilania zdarzy się w czasie wykonywania operacji zapisu, uszkodzeniu ulegają zapisywane wtedy dane. Taka sytuacja symulowana jest poprzez wyzerowanie bloków w ostatnim wykonanym zleceniu zapisu. Ostatni punkt odpowie na pytanie, czy sterowniki systemów plików zachowują spójność metadanych.

4.2.1. Wyniki

Kolejne tabele zawierają liczbę operacji, które nie były widoczne po symulowanej awarii. W przypadku operacji na katalogach (`mkdir` i `rmdir`) wykonane było 1057 wywołań, natomiast dla operacji na plikach (`create`, `unlink`, `rename` i `append`) było 4096 wywołań. Testy przeprowadziłem dla systemów plików ext3, ReiserFS w wersji 3, JFS i XFS.

Tabele od 4.1 do 4.4 zawierają wyniki testów odcinających dostęp do urządzenia po wykonaniu ostatniej operacji. Tabele od 4.5 do 4.8 zawierają wyniki testów destrukcyjnych, w których wraz z odcięciem dostępu do urządzenia były uszkadzane sektory należące do ostatniej wykonanej operacji zapisu. Wyniki testów ulegały pewnym, drobnym wahaniom w kolejnych uruchomieniach, jednak różnice były na tyle małe, że można uznać podane wyniki za reprezentatywne.

Tylko systemy plików ext3 i XFS mogą być zamontowane w trybach *sync* i *dirsync*, dlatego ReiserFS i JFS zostały pominięte w testach wykorzystujących te tryby pracy. Dodatkowo testy destrukcyjne dla systemu plików JFS okazały się niewykonalne, ponieważ sterownik dla JFS posiada poważne błędy uniemożliwiające zamontowanie uszkodzonego systemu plików.

Wykonane testy pokazały, że sprawdzane systemy plików zachowują się zgodnie z oczekiwaniami. Flaga `O_SYNC` powinna zapewnić, że wszystkie zapisy do pliku znajdą się na nośniku zanim program zapisujący otrzyma informację o wykonaniu zadania. Jak widać, tylko w przypadku destrukcyjnych testów gubione były pojedyncze zapisy. Zadaniem funkcji `fsync()` jest wymuszenie zapisu wszystkich danych i metadanych związanych z deskryptorem pliku lub katalogu, podanym jako parametr w wywołaniu. Tylko na systemie plików JFS ta funkcja nie zapewniała spójności metadanych, na pozostałych użycie `fsync()` daje pewność, że dane nie zostaną utracone po awarii. Widać też, że nie ma potrzeby korzystania z `O_SYNC` i `fsync()` jeżeli system plików zamontowany jest w trybie *sync*. Użycie tego trybu zapewnia spójność danych w przypadku awarii.

Znaczne ilości danych i metadanych mogą ulec uszkodzeniu tylko w przypadku, gdy nie ma żadnego mechanizmu wymuszającego zapisy, co widać w tabelach 4.1, 4.5 i dla operacji `append` w tabelach 4.3 oraz 4.7. Przy czym system plików XFS okazuje się być najmniej podatnym na awarie, natomiast ReiserFS najbardziej podatnym. Jeżeli jest użyty mechanizm wymuszający zapisy, wtedy nawet przy wymazaniu ostatnich zapisów tracone są pojedyncze informacje (tabele od 4.5 do 4.8).

	mkdir	rmdir	create	unlink	rename	append
operacje wykonane z flagą `O_SYNC`						
ext3	181	1057	1659	1823	1243	0
XFS	2	10	5	8	58	0
JFS	2	4	3	6	4	0
ReiserFS	1057	1057	4096	4096	4096	0
operacje potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
JFS	2	4	3	6	4	0
ReiserFS	0	0	0	0	0	0
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
JFS	2	4	3	6	4	0
ReiserFS	0	0	0	0	0	0

Rysunek 4.1: Liczba operacji niewidocznych po awarii

	mkdir	rmdir	create	unlink	rename	append
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje wykonane z flagą `O_SYNC`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0

Rysunek 4.2: Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync*

	mkdir	rmdir	create	unlink	rename	append
ext3	0	0	0	0	0	4096
XFS	0	0	0	0	0	579
operacje wykonane z flagą `O_SYNC`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0

Rysunek 4.3: Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *dirsync*

	mkdir	rmdir	create	unlink	rename	append
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje wykonane z flagą `O_SYNC`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	0	0	0	0	0	0
XFS	0	0	0	0	0	0

Rysunek 4.4: Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync* i *dirsync*

	mkdir	rmdir	create	unlink	rename	append
operacje wykonane z flagą `O_SYNC`						
ext3	1057	0	4096	4096	2667	1
XFS	19	24	63	38	188	0
ReiserFS	1057	1057	4096	4096	4096	1
operacje potwierdzone funkcją `fsync()`						
ext3	1	1	0	1	1	2
XFS	1	1	2	2	2	2
ReiserFS	1	1	1	1	1	1
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	1	1	0	1	1	1
XFS	1	1	2	2	2	0
ReiserFS	1	1	1	1	1	1

Rysunek 4.5: Liczba operacji niewidocznych po awarii i uszkodzeniu ostatniego zapisu

	mkdir	rmdir	create	unlink	rename	append
ext3	0	0	1	0	1	0
XFS	1	2	2	2	2	0
operacje wykonane z flagą `O_SYNC`						
ext3	1	0	1	0	1	1
XFS	1	2	2	2	2	0
operacje potwierdzone funkcją `fsync()`						
ext3	0	0	1	0	0	0
XFS	1	2	2	2	2	0
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	0	0	1	0	0	1
XFS	1	2	2	2	2	0

Rysunek 4.6: Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync* i uszkodzeniu ostatniego zapisu

	mkdir	rmdir	create	unlink	rename	append
ext3	1	1	1	0	1	4096
XFS	1	2	2	2	2	552
operacje wykonane z flagą `O_SYNC`						
ext3	1	1	1	0	1	1
XFS	0	2	2	2	2	0
operacje potwierdzone funkcją `fsync()`						
ext3	0	0	1	0	0	1
XFS	1	2	2	2	2	2
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	0	0	1	0	0	1
XFS	1	2	2	2	2	0

Rysunek 4.7: Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *dirsync* i uszkodzeniu ostatniego zapisu

	mkdir	rmdir	create	unlink	rename	append
ext3	1	0	1	0	1	0
XFS	1	2	2	0	2	0
operacje wykonane z flagą `O_SYNC`						
ext3	1	0	1	0	1	1
XFS	1	2	2	2	2	0
operacje potwierdzone funkcją `fsync()`						
ext3	0	0	1	0	0	0
XFS	1	2	2	2	2	0
operacje wykonane z flagą `O_SYNC` i potwierdzone funkcją `fsync()`						
ext3	0	0	1	0	0	1
XFS	1	2	2	2	2	0

Rysunek 4.8: Liczba operacji niewidocznych po awarii na systemie plików zamontowanym w trybie *sync* i *dirsync* i uszkodzeniu ostatniego zapisu

Rozdział 5

Podsumowanie

Celem pracy była implementacja sterownika symulującego różne rodzaje awarii urządzeń blokowych. Sterownik pozwala symulować różne zachowania urządzeń pamięci masowych w przypadku uszkodzenia, takie jak błędy wejścia-wyjścia i przekłamywanie danych w odczytywanych lub zapisywanych sektorach. Wykorzystanie podsystemu *device mapper* do implementacji sterownika pozwala używać go na dowolnym sprzęcie, na którym można uruchomić system Linux. Uważam, że przedstawione rozwiązanie spełnia założenia postawione w punkcie 3.1. Dużym ułatwieniem przy implementacji były dobrze zdefiniowane i przejrzyste interfejsy podsystemów jądra Linuksa, w tym podsystemu *device mapper*.

Wykonane przy pomocy sterownika testy transakcyjności systemów plików pokazały, jak dostępne w Linuksie systemy plików z dziennikowaniem zachowują się w przypadku awarii typu odcięcie zasilania. Z testów wynika, że najpopularniejsze systemy plików, czyli ext3 i XFS, zachowują się zgodnie z oczekiwaniami i zachowują spójność metadanych. Równie dobrze wypadł w testach ReiserFS, ale brak wsparcia dla synchronicznej pracy znacznie utrudnia wykorzystanie go w sytuacjach, w których priorytetem jest bezpieczeństwo danych. Dużo gorzej wypadł JFS, który gubił operacje pomimo wymuszania zapisów. Także błędy w sterowniku dyskwalifikują go w poważnych zastosowaniach.

Funkcjonalność sterownika można rozszerzyć o wsparcie dla modułów sterujących jego zachowaniem na podstawie analizy zawartości bloków w czasie wykonywania operacji wejścia-wyjścia. Pozwoliłoby to, przykładowo, testować odporność na awarie systemów plików w przypadku uszkodzenia dynamicznych struktur danych. Innym możliwym rozszerzeniem jest wykorzystanie mechanizmu *blktrace* do komunikacji z programami działającymi w przestrzeni użytkownika. Przed wykonaniem operacji sterownik wysyłałby komunikat z jej opisem, a następnie oczekiwał na odpowiedź mówiącą w jaki sposób operację obsłużyć.

Dodatek A

Zawartość płyty dołączonej do pracy

Na płycie znajdują się:

- elektroniczna wersja pracy,
- kod źródłowy sterownika dm-fault,
- kod źródłowy systemu Linux 2.6.26,
- biblioteka libdmfault,
- skrypty testujące poprawność działania sterownika,
- skrypty i programy testujące system plików pod względem odporności na awarie typu odcięcie zasilania,
- środowisko testowe dla emulatora VirtualBox.

Bibliografia

[Axboe02] Jens Axboe, Suparna Bhattacharya, *Notes on the Generic Block Layer Rewrite in Linux 2.5*, http://lxr.linux.no/linux+v2.6.26/Documentation/block/biodoc.txt

[BGPS07] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, Jiri Schindler, *An Analysis of Latent Sector Errors in Disk Drives*, International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'07), San Diego, California, June 2007

[BGSAA08] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, *An Analysis of Data Corruption in the Storage Stack*, 6th USENIX Conference on File and Storage Technologies, San Jose, CA, February 2008

[DMIO] *Device mapper I/O services*, http://lxr.linux.no/linux+v2.6.26/Documentation/device-mapper/dm-io.txt

[DM] *Device-mapper Resource Page*, http://sources.redhat.com/dm/

[DS40] The DiskSim Simulation Environment (v4.0), http://www.pdl.cmu.edu/DiskSim/, 26 września 2008

[Ganger95] Gregory R. Ganger, *System-Oriented Evaluation of I/O Subsystem Performance*, Doctoral Dissertation, University of Michigan, 1995

[GvI05] Jim Gray, Catharine van Ingen, *Empirical Measurements of Disk Failure Rates and Error Rates*, Microsoft Research Technical Report MSR-TR-2005-166, December 2005

[GGAAL08] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Ben Liblit, *EIO: Error Handling is Occasionally Correct*, 6th USENIX Conference on File and Storage Technologies, San Jose, CA, February 2008

[JiHuZ08] Weihang Jiang, Chongfeng Hu, and Yuanyuan Zhou, *Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics*, 6th USENIX Conference on File and Storage Technologies, San Jose, CA, Feb. 26-29, 2008

[LinuxKernel05] Robert Love, *Linux Kernel Development Second Edition*, Sams Publishing, January 12, 2005

[PWB07] Eduardo Pinheiro, Wolf-Dietrich Weber, Luiz André Barroso, *Failure Trends in a Large Disk Drive Population*, 5th USENIX Conference on File and Storage Technologies, San Jose, CA, Feb. 14-16, 2007

[ViPr06] Vijayan Prabhakaran, *IRON File Systems*, Doctoral Dissertation, University of Wisconsin-Madison, June 2006

[Tal99] Nisha Darshi Talagala, *Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks*, Doctoral Dissertation, University of California, Berkeley, 1999

[Schlosser03] Steven W. Schlosser, Gregory R. Ganger, *MEMS-based storage devices and standard disk interfaces: A square peg in a round hole?*, 3rd USENIX Conference on File and Storage Technologies, March 2004

[SG07] Bianca Schroeder, Garth A. Gibson, *Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?*, 5th USENIX Conference on File and Storage Technologies, San Jose, CA, Feb. 14-16, 2007

[SSFZ99] Franklin E. Sorenson, Elizabeth S. Sorenson, J. Kelly Flanagan, Heng Zhou, *A System-Assisted Disk I/O Simulation Technique*, 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, October 1999

[Szlufik06] Marcin Szlufik, *Implementacja macierzy RAID z konfigurowalnym stopniem nadmiarowości dla systemu Linux*, praca magisterska na kierunku informatyka, Uniwersytet Warszawski, Wydział MIM, Lipiec 2006

[Worthington94] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, *Scheduling Algorithms for Modern Disk Drives*, ACM Sigmetrics Conference, May, 1994