**Uniwersytet Warszawski**
Wydział Matematyki, Informatyki i Mechaniki

**Jerzy Szczepkowski**
Nr albumu: 189202

# Asynchroniczne wejście-wyjście w systemie Linux

**Praca magisterska**
**na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem
**dr Janiny Mincer-Daszkiewicz**
Instytut Informatyki

Wrzesień 2004

Pracę przedkładam do oceny

Data Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data Podpis kierującego pracą:

## Streszczenie

W niniejszej pracy przedstawione są zagadnienia związane z operacjami asynchronicznego odczytu i zapisu realizowanymi przez systemy operacyjne. Opisana jest implementacja operacji asynchronicznych w jądrze systemu Linux 2.6 oraz zaprezentowane są łaty na jądro Linuksa rozszerzające ich funkcjonalność oraz poprawiające wydajność.



## Słowa kluczowe

Linux 2.6, asynchroniczne wejście-wyjście, operacje asynchroniczne, epoll, poll,
łata na jądro, implementacja systemu plików

## Klasyfikacja tematyczna

D. Software
D.4 Operating Systems
D.4.3 File Systems Management

# Spis treści

# Spis rysunków

# Spis tabel

# Rozdział 1

# Wprowadzenie

W odróżnieniu od operacji synchronicznej, wykonanie operacji asynchronicznego wejścia-wyjścia (ang. *Asynchronous Input/Output*, w skrócie AIO), z punktu widzenia zleceniodawcy, składa się z dwóch faz:

1. zlecenia operacji,
2. pobrania informacji zwrotnej, że operacja wykonała się.

Implementacja AIO odpowiada za wykonanie zleconego odczytu (zapisu) pomiędzy fazą pierwszą a drugą oraz za przekazanie informacji o jej zakończeniu. Zlecający proces może w międzyczasie wykonywać swoje zadania — odczyt (zapis) wykona się współbieżnie z jego ścieżką wykonania. Podstawowymi zaletami operacji asynchronicznych są:

- możliwość wykonywania innych zadań współbieżnie z operacjami odczytu (zapisu),
- umożliwienie lepszego szeregowania żądań do urządzeń wejścia-wyjścia.

Coraz częściej skłaniają one projektantów wysokowydajnych aplikacji (takich jak systemy obsługi baz danych czy serwery sieciowe), wykonujących wiele operacji wejścia-wyjścia, do używania w nich AIO.

AIO jest obecnie obsługiwane przez wszystkie liczące się systemy operacyjne. Operacje asynchroniczne zostały także dodane do jądra systemu Linux, począwszy od wersji 2.5. Niestety, ubogo prezentuje się Linuksowy interfejs, za pomocą którego proces może otrzymać informację zwrotną o zakończeniu zleconych operacji. Jest on wyspecjalizowany (pozwala na otrzymywanie informacji jedynie o zakończeniu operacji asynchronicznych) i nie zintegrowany z żadnym innym, bardziej ogólnym mechanizmem powiadamiania (pozwalającym np. na uzyskanie informacji o gotowości deskryptorów plików do odczytu-zapisu).

W niniejszej pracy przedstawiono zagadnienia związane z operacjami asynchronicznymi, ze szczególnym naciskiem na ich realizację w systemie Linux. Zaproponowano również dwie łaty modyfikujące jądro Linuksa. Rozszerzają one funkcjonalność mechanizmu powiadamiania *event-poll* poprzez umożliwienie oczekiwania na zdarzenia zakończenia operacji asynchronicznego wejścia-wyjścia.

*Event-poll* jest wydajniejszą wersją klasycznej funkcji systemowej `poll` w Linuksie. Przy korzystaniu z *event-poll* proces na początku dokonuje rejestracji deskryptorów (funkcja `epoll_ctl`), którymi jest zainteresowany, a później oczekuje na ich gotowość do odczytu lub zapisu (funkcja `epoll_wait`). Poprzez wprowadzenie dwóch faz, rejestracji i oczekiwania, nie ma konieczności skanowania całego zestawu deskryptorów przy każdym czekaniu, tak jak ma to miejsce w przypadku klasycznego `poll`. Dzięki wprowadzonym w łatach modyfikacjom

pojedynczy wątek, np. serwera sieciowego, może równocześnie oczekiwać na zakończenie zleconych operacji asynchronicznych i gotowość gniazda do odczytu (nadejście nowego zlecenia od klienta).

Jedna z opracowanych łat ma również na celu poprawę wydajności obsługi AIO co, jak dowiedziono przez przeprowadzone testy, udało się osiągnąć.

W trakcie prac nad łatami, autorowi udało się znaleźć dwa błędy w jądrze Linuksa. Jeden z nich, skutkujący wyciekiem pamięci, był dość poważny i naruszał bezpieczeństwo całego systemu. Obydwa zostały naprawione, a korygujące je łaty stały się częścią oficjalnej dystrybucji jądra.

## 1.1. Struktura pracy

W rozdziale 2 zdefiniowano czym są operacje asynchroniczne, podano ich zastosowania oraz cechy, jakimi charakteryzują się ich implementacje.

W rozdziale 3 zaprezentowano i porównano interfejsy do obsługi AIO: posiksowy oraz udostępniane przez systemy operacyjne Linux, Windows i FreeBSD.

W rozdziale 4 opisano implementację mechanizmów jądra Linuksa, w których wprowadzono modyfikacje: operacji asynchronicznych oraz *event-poll*.

Rozdział 5 zawiera opis łat rozszerzających funkcjonalność jądra Linuksa, stworzonych w ramach niniejszej pracy.

W rozdziale 6 przedstawione są wyniki testów porównujących działanie oryginalnego jądra systemu Linux z wersjami zmodyfikowanymi.

W rozdziale 7 opisane są błędy w jądrze Linuksa, które udało się autorowi znaleźć i naprawić.

Rozdział 8 zawiera podsumowanie pracy.

W dodatku A opisana jest zawartość dołączonej do pracy płyty CD z opracowanymi łatami.

# Rozdział 2

# Zagadnienie asynchronicznego wejścia-wyjścia

## 2.1. Wstęp

W rozdziale tym opisano czym jest asynchroniczna operacja wejścia-wyjścia. Podano także przykłady zastosowania operacji asynchronicznych oraz główne cechy jakimi mogą charakteryzować się ich implementacje.

## 2.2. Charakterystyka AIO

Z punktu widzenia procesu wykonanie operacji asynchronicznej składa się z dwóch faz:

1. zlecenia wykonania operacji asynchronicznej,
2. otrzymania informacji zwrotnej, że zlecona operacja asynchroniczna wykonała się (ew. że z jakiegoś powodu nie udało się jej wykonać).

Pomiędzy fazą pierwszą a drugą operacja wykonuje się współbieżnie ze ścieżką wykonania zlecającego procesu. Szczególnie istotne jest by pierwsza faza — zlecenie wykonania operacji — realizowana zazwyczaj przez wywołania odpowiednich funkcji systemowych lub bibliotecznych (takich jak operacje `aio_read` we FreeBSD, `aio_submit` pod Linuksem, `ReadFileEx` w systemach Windows) była jak najmniej blokująca. Niepożądana jest sytuacja, w której proces w trakcie zlecania mógłby zasnąć *na dłużej* w oczekiwaniu na jakiś zasób.

Implementacje drugiej fazy operacji przewidują zazwyczaj kilka sposobów otrzymywania informacji zwrotnej. Proces może *odpytywać* (sprawdzać w sposób nieblokujący aktualny stan operacji). Może oczekiwać na wykonanie się jednej (lub kilku) spośród zbioru zleconych operacji. Po wykonaniu operacji do procesu może zostać wysłany sygnał lub wstawione odpowiednie zdarzenie do wskazanej kolejki zdarzeń. Możliwe jest wreszcie asynchroniczne wywołanie wskazanej procedury z przestrzeni procesu lub uruchomienie wewnątrz niego nowego wątku. Problem powiadamiania będzie jeszcze poruszany w rozdziale 3, przy przeglądzie realizacji AIO w różnych systemach operacyjnych.

Aby umożliwić działanie AIO, stawiane jest wymaganie by pomiędzy fazą pierwszą i drugą proces zlecający w żaden sposób nie modyfikował obszarów pamięci, których dotyczy dane zlecenie [posix]. W przypadku wykonania jakichkolwiek modyfikacji rezultat operacji asynchronicznej jest nie określony.

Standard POSIX określa jeszcze kilka dodatkowych wymagań na implementację AIO, m.in.:

- Możliwość zlecenia wykonania wielu operacji na wielu deskryptorach za pomocą pojedynczego wywołania.
- Możliwość odwoływania (kasowania) zleceń.
- Możliwość oczekiwania na zdarzenie zakończenia operacji w połączeniu z oczekiwaniem na zdarzenia innych typów.

## 2.3. Zastosowania AIO

Najistotniejszą grupę użytkowników AIO [aio, asl25, posix, coek] stanowią aplikacje wykonujące wiele operacji wejścia-wyjścia. Dzięki operacjom asynchronicznym możliwe jest zminimalizowanie czasu traconego na kosztowne wywołania funkcji systemowych poprzez ich nieblokowanie oraz możliwość zlecania wielu operacji za pomocą jednego wywołania. Do takich aplikacji można zaliczyć m. in.:

- serwery sieciowe,
- serwery pośredniczące (ang. *proxy*),
- systemy obsługi baz danych dużego rozmiaru.

Możemy wyobrazić sobie również szereg innych zastosowań AIO, np. przez aplikację, która na potrzeby prowadzenia rejestru zdarzeń wykonuje wiele nisko-priorytetowych operacji zapisu.

### 2.3.1. Przykład serwera sieciowego

Wyobraźmy sobie, że chcemy stworzyć serwer sieciowy, mogący obsługiwać dużą liczbę klientów. Za [c10k] prześledźmy strategie, jakie możemy zastosować do realizacji operacji wejścia-wyjścia:

#### A. Dla każdego klienta serwer tworzy osobny wątek

To rozwiązanie wydaje się najprostsze do zaimplementowania. Do obsługi każdego z klientów tworzony jest osobny wątek, który będzie wykonywać blokujące operacje odczytu-zapisu. Niestety przy dużej liczbie klientów rozwiązanie to może powodować zbyt duże narzuty na przeszeregowanie oraz pamięć (każdy wątek posiada własny stos wykonania) i przez to znacznie tracić na wydajności.

#### B. Jeden wątek serwera obsługuje wielu klientów poprzez odpytywanie deskryptorów i nieblokujące operacje wejścia-wyjścia

Serwer wykonuje operację wejścia-wyjścia dopiero wtedy, gdy zostanie poinformowany o gotowości deskryptora i operacja ta będzie mogła zostać wykonana bez blokowania.

W przypadku standardowych funkcji `select` i `poll` każde wywołanie powoduje konieczność przesyłania oraz skanowania całego zbioru deskryptorów, na które oczekujemy. Rozwiązanie to staje się nieefektywne gdy deskryptorów tych jest dużo. Dodatkowo powinno uwzględnić się możliwość oczekiwania przez wiele wątków na tym samym zestawie deskryptorów.

Z tego powodu dzisiejsze systemy operacyjne dostarczają bardziej zaawansowane rozwiązania pozbawione tych wad. Sprowadzają się one do rozdzielenia rejestracji deskryptorów,

którymi jesteśmy zainteresowani, od oczekiwania na ich gotowość. System FreeBSD pozwala na dowiadywanie się o gotowości deskryptora poprzez mechanizm kolejek zdarzeń (ang. *kqueue*). W Linuksie można robić to za pomocą rodziny funkcji systemowych `epoll` (ang. *event-poll*).

**Sterowanie stanem/sterowanie zmianą stanu** Rozważmy wywołanie funkcji odpytującej dla zarejestrowanego zestawu deskryptorów. Możliwe są dwie semantyki gotowości deskryptora:

1. **Sterowana stanem (ang. *level-triggered*)**

   Przekazywane są te deskryptory, które są gotowe w momencie wywołania funkcji (standardowo, tak działa klasyczny `select` i `poll`).

2. **Sterowana zmianą stanu (ang. *edge-triggered*)**

   Przekazywane są te deskryptory, które od momentu ostatniego wywołania funkcji przeszły ze stanu niegotowy do gotowy. Zalety takiego podejścia widać w przypadku, gdy na zarejestrowanym zestawie oczekuje kilka wątków. O gotowości deskryptora zostanie wtedy poinformowany tylko pierwszy z nich. Własność ta nazwana jest *jednostrzałowością* (ang. *one-shot*).

Zwróćmy również uwagę na fakt, że przytoczone mechanizmy odpytywania działają tylko dla deskryptorów niektórych typów: łączy (ang. *pipe*), gniazd (ang. *socket*), kolejek FIFO. W szczególności nie działają dla plików dyskowych (systemy plików EXT2, FAT).

#### C. Jeden wątek obsługuje wielu klientów poprzez operacje asynchroniczne

Zlecenie operacji AIO jest nieblokujące, jeden proces może więc wykonywać równocześnie bardzo wiele takich operacji. Przeszkodę stanowi fakt, że operacje asynchroniczne nie są wspierane przez wszystkie systemy operacyjne, a ich implementacje bywają niekompletne (np. nie działają dla wszystkich typów plików). Brakuje również ujednoliconego interfejsu dla AIO.

## 2.4. Implementacje AIO

### 2.4.1. Poziom realizacji

Realizacja operacji asynchronicznych może odbywać się na różnych poziomach [aio, asl25]:

- Operacje są w całości realizowane przez bibliotekę z poziomu użytkownika, np. za pomocą wątków i zwykłych operacji synchronicznych, tak jak w przypadku biblioteki glibc.
- Operacje wykonują się na poziomie systemu operacyjnego. Interfejs stanowią funkcje systemowe (FreeBSD, Windows, Linux od wersji 2.5).
- Podejście hybrydowe — operacje wykonywane są przez bibliotekę z poziomu użytkownika, lecz z częściowym wsparciem ze strony systemu operacyjnego. Przykładem może być biblioteka KAIO z łatą dla jądra Linuksa wydana przez SGI.

Asynchroniczne wejście-wyjście realizowane w całości na poziomie użytkownika jest zdecydowanie mniej wydajne od realizacji wspieranych przez jądro i nie nadaje się do tworzenia w oparciu o nie wydajnych aplikacji. W niniejszej pracy będziemy zajmować się realizacjami z poziomu systemu operacyjnego.

### 2.4.2. Operacje synchroniczne i asynchroniczne

Wyobraźmy sobie system operacyjny udostępniający zarówno operacje synchroniczne, jak i asynchroniczne [asl25]. Operację synchroniczną możemy uznać za szczególny przypadek operacji asynchronicznej — proces zleca wykonanie operacji asynchronicznej oraz czekanie na jej zakończenie. Dzięki temu można wszystkie operacje na plikach zaimplementować jako asynchroniczne, tak jak jest to zrobione w systemie WindowsNT.

Podejście takie ma zarówno wady, jak i zalety. Można łatwo unikać dublowania kodu o bardzo podobnej funkcjonalności. Z drugiej strony utrudniona jest naturalna optymalizacja, polegająca w przypadku operacji synchronicznej na zminimalizowaniu czasu oczekiwania na pojedyncze wykonanie (ang. *latency*), a w przypadku operacji asynchronicznych na zmaksymalizowaniu ich przepustowości (ang. *throughput*).

Alternatywą są odrębne mechanizmy dla operacji synchronicznych i asynchronicznych, tak jak ma to miejsce w systemach FreeBSD czy Linux (zagadnienie to w odniesieniu do jądra Linuksa będzie dyskutowane szerzej w rozdziale 4).

Rozróżnienie to może mieć również odbicie w dostarczanym przez system operacyjny interfejsie. W Windows zarówno operację odczytu synchronicznego, jak i asynchronicznego można zlecać za pomocą tej samej funkcji `ReadFile`, podczas gdy standard POSIX określa całkowicie odrębne funkcje dla operacji synchronicznych i asynchronicznych [posix].

### 2.4.3. Pośrednie kopiowanie

W przypadku plików służących do komunikacji sieciowej (np. gniazd) istotną cechą realizacji AIO, podnoszoną w wielu publikacjach, jest brak pośredniego kopiowania (ang. *zero-copy*). Polega to na bezpośrednim przenoszeniu danych pomiędzy przestrzenią adresową użytkownika a urządzeniem (np. kartą sieciową).

Należy zachować ostrożność przy próbie przeniesienia idei braku pośredniego kopiowania na inne urządzenia, takie jak dyski. Kłóci się ona z realizowanym przez system operacyjny buforowaniem bloków dyskowych. Operacja asynchroniczna wykonywana za pośrednictwem bufora jest operacją *z pośrednim kopiowaniem*. W jądrze Linuksa 2.6 operacje asynchroniczne zaimplementowane są jedynie dla plików otwartych z wyłączonym buforowaniem (z flagą `O_DIRECT`). Zakłada się, że ew. buforowanie powinno być realizowane po stronie aplikacji.

### 2.4.4. Nieblokowanie

Implementacja AIO powinna gwarantować możliwie szybki powrót z funkcji zlecającej operację [aio]. Jednak w praktyce nie da się w całości uniknąć oczekiwania. Dostępnym implementacjom zdarza się oczekiwać np. w następujących sytuacjach (Linux 2.6):

1. uzyskiwanie dostępu do obszarów pamięci,
2. alokowanie przez jądro pamięci na rekord kontrolny dla operacji,
3. oczekiwanie na semaforze i-węzła.

Możliwa jest również sytuacja, w której przy dużym obłożeniu, system operacyjny nie będzie w stanie zlecić kolejnej operacji z powodu przepełnienia bufora. Funkcja w takiej sytuacji powinna przekazać błąd [EAGAIN]. Warunek na to (np. określający maksymalną liczbę oczekujących operacji w systemie operacyjnym) powinien być dokładnie określony.

Zgodnie ze standardem POSIX funkcja powinna zakończyć swoje działanie dopiero wtedy, gdy zlecenie zostanie zainicjowane lub zakolejkowane do pliku lub urządzenia, nawet wtedy, gdy nie można zrobić tego natychmiast [posix].

Natychmiastowy powrót z funkcji zlecającej i zgłaszanie błędu [EAGAIN] (np. w każdej z sytuacji 1 - 3) może okazać się w praktyce mało konstruktywne i prowadzić do konieczności stosowania aktywnego zlecania (wielokrotnego powtarzania zlecenia) przez wywołujący proces.

### 2.4.5. Powiadamianie

#### Powiadamianie poprzez odpytywanie/oczekiwanie

Zdarzenia związane z zakończeniem wykonania operacji AIO gromadzone są w jakiegoś typu buforach [aio] (najczęściej w kolejkach). Funkcje systemowe umożliwiające wątkowi otrzymanie informacji o tym, jak zakończyło się jego zlecenie, powiązane są zazwyczaj z tymi właśnie buforami. Funkcje te umożliwiają również podanie maksymalnego czasu oczekiwania oraz pozwalają na oczekiwanie wielu wątkom na tym samym buforze.

1. **Kontekst AIO**

   Rozwiązanie z systemu Linux. Przed rozpoczęciem wykonywania operacji AIO trzeba utworzyć dla niej kontekst (strukturę jądra zawierającą kolejkę zdarzeń). Każda operacja wykonuje się w jednym, określonym kontekście, natomiast z jednym kontekstem może być związane wiele operacji. Proces może oczekiwać w kontekście na zdarzenia ukończenia wykonujących się w nim operacji AIO.

2. **Uogólniona kolejka zdarzeń** (*kqueue*)

   Takie rozwiązanie zastosowane jest w systemie FreeBSD [lemon]. Kqueue jest ogólnym mechanizmem powiadamiania. Pozwala procesowi na zgłoszenie swojego zainteresowania wybranymi zdarzeniami, które mogą zachodzić na wybranych deskryptorach plików. W momencie wystąpienia takiego zdarzenia, informacja o nim zostaje wstawiona do kolejki. Zdarzeniem takim mogą być m.in. pomyślne zakończenie operacji asynchronicznej lub zgłoszenie gotowości deskryptora do odczytu (zastępuje funkcje `poll`).

3. **IOCP** (*IO Completion Port*)

   Występuje w systemach Windows. Jest to rodzaj kolejki zdarzeń, lecz tylko dla operacji AIO. Deskryptor możemy zarejestrować w IOCP, do którego wysyłane będą zdarzenia informujące o wykonanych na nim operacjach. Pod określonym numerem portu może równocześnie oczekiwać wiele wątków tego samego procesu. Można ustalać tzw. stopień współbieżności portu — maksymalną liczbę wykonujących się wątków związanych z portem (zalecane jest wybranie stopnia równego liczbie procesorów w komputerze).

#### Powiadamianie asynchroniczne [aio]

1. **Sygnał**

   Proces zlecający może związać operacje AIO z sygnałem o określonej wartości, który zostanie do niego wysłany w momencie jej zakończenia (por. p. 3.2 i 3.5).

2. **Wątek**

   Po zakończeniu operacji zostanie uruchomiany nowy wątek w ramach procesu [posix].

3. **APC** (*Asynchronous Procedure Call*)

   Mechanizm występujący w systemach Windows. W momencie zakończenia operacji AIO do wątku, który ją zlecał może zostać dołączona procedura asynchroniczna. Wywoła się ona w jego przestrzeni adresowej.

### 2.4.6. Szeregowanie operacji

Od implementacji AIO zależy kolejność, w jakiej wykonywane będą zlecone operacje [aio, posix]. Naturalne jest wybranie takiego ciągu realizacji, przy którym optymalizowana jest przepustowość lub czas wykonanie pojedynczego zlecenia.

W pewnym stopniu kolejność wykonywania operacji asynchronicznych może być określana przez samego zlecającego. Standard POSIX przewiduje dwie takie sytuacje (zachodzące po otwarciu pliku z określoną flagą):

1. **SIO** (ang. *Synchronized I/O*)

   Jeśli jest wiele operacji odnoszących się do tego samego pliku, to będą one realizowane dokładnie w tej kolejności, w jakiej zostały zlecone.

2. **PIO** (ang. *Prioritized I/O*)

   Każde zlecenie posiada swój priorytet. Na priorytet składają się dwa elementy — priorytet procesu używany przy jego szeregowaniu oraz modyfikator podawany w zleceniu (dokładnie: priorytet zlecenia jest ich różnicą, w ten sposób możemy jedynie spowolnić wykonanie operacji). Przy wyborze preferowane są zlecenia o wyższym priorytecie, szczegóły można znaleźć w [posix], *System Interfaces*, Issue 6, p. 2.8.2.

Funkcjonalności SIO i PIO określone są w specyfikacji POSIX jako opcjonalne.

# Rozdział 3

# Przegląd interfejsów AIO

## 3.1. Wstęp

W rozdziale tym zaprezentowano i porównano interfejsy do obsługi operacji asynchronicznych.

## 3.2. POSIX

Standard POSIX wymaga, by każda operacja asynchroniczna była określana przez rekord kontrolny `aicob` (ang. *AIO control block*). Rekord ten powinien zawierać następujące pola:

- `aio_lio_opcode` — kod operacji, którą chcemy wykonać (LIO_READ — odczyt, LIO-_WRITE — zapis),
- `aio_reqprio` — priorytet operacji,
- `aio_fildes` — deskryptor pliku, na którym ma zostać wykonana operacja,
- `volatile void *aio_buf` — bufor pamięci użytkownika, do którego będą zapisywane dane (lub odczytywane dane do zapisu),
- `aio_nbytes` — liczba bajtów, które chcemy odczytać (zapisać),
- `aio_offset` — pozycja w pliku, od której chcemy rozpocząć operacje odczytu (zapisu),
- `struct sigevent aio_sigevent` — wartość sygnału, który po zakończeniu operacji ma być wysłany do zleceniodawcy.

Gdy wypełnimy rekord kontrolny, wykonanie operacji asynchronicznej zlecamy za pomocą funkcji:

```
int aio_read(struct aiocb *aiocbp)   // dla odczytu
```

lub

```
int aio_write(struct aiocb *aiocbp)  // dla zapisu.
```

Gdy zlecenie zakończy się pomyślnie, funkcje przekazują `0`; gdy zlecenie nie powiedzie się, przekazywane jest `-1` i ustawiany kod błędu na odpowiednią wartość.

Do oczekiwania na zakończenie jednej spośród wielu operacji AIO służy funkcja

```
int aio_suspend(struct aiocb *list[], int nent,
                struct timespec *timeout).
```

Jako parametr przyjmuje ona tablicę `list` wskaźników do rekordów kontrolnych zleconych operacji, jej rozmiar `nent` i czas `timeout` przez jaki chcemy maksymalnie oczekiwać. Funkcja przekazuje `0` w przypadku gdy zakończyła się przynajmniej jedna z podanych operacji.

By sprawdzić, w jakim aktualnie stanie znajduje się zlecona operacja należy wywołać funkcję

```
int aio_error(struct aiocb *aiocbp),
```

której parametrem jest wskaźnik do rekordu kontrolnego aiocb. W przypadku gdy operacja zakończyła się pomyślnie, przekazywane jest 0; gdy operacja trwa — błąd [`EINPROGRESS`]; w przeciwnym wypadku kod błędu, którym się zakończyła.

Gdy operacja asynchroniczna została zakończona, za pomocą funkcji

```
size_t aio_return(struct aiocb *aiocbp)
```

możemy dowiedzieć się, jaką wartość przekazała (np. dla operacji odczytu będzie to liczba bajtów, które udało się odczytać). Funkcję tę możemy wywołać tylko wówczas, gdy sprawdziliśmy poprzez `aio_error`, że operacja jest już zakończona i nie więcej niż raz dla każdej operacji. W przeciwnym wypadku rezultat jej działania nie jest określony.

Za pomocą funkcji

```
int aio_cancel(int filedes, struct aiocb *aiocbp)
```

możemy próbować odwołać operacje asynchroniczną, do której rekordu kontrolnego podajemy wskaźnik lub wszystkie operacje, wykonujące się na określonym deskryptorze pliku.

Funkcja

```
int lio_listio(int mode, struct aiocb *list[],
               int nent, struct sigevent *restrict sig)
```

umożliwia zlecenie wielu operacji za pomocą pojedynczego wywołania. Jako argumenty przyjmuje tablicę `list` wskaźników do rekordów kontrolnych oraz tryb `mode` w jakim mają być wykonane operacje (synchronicznie lub asynchronicznie).

### 3.2.1. Wady i zalety

Proponowany przez POSIX interfejs nie jest wydajny. Zwróćmy uwagę na to, że obsługa operacji asynchronicznej składa się aż z czterech wywołań funkcji: `aio_read`/`aio_write`, `aio_suspend`, `aio_error`, `aio_return`. Może okazać się to szczególnie nieefektywne, gdy są one realizowane za pomocą kosztownych wywołań systemowych, których liczbę pragniemy minimalizować. Optymalnie powinna istnieć możliwość obsługi operacji asynchronicznej za pomocą dwóch wywołań: do zlecania i do otrzymywania informacji zwrotnej o zakończeniu lub błędzie, w połączeniu z możliwością oczekiwania.

Niewydajny jest również zaproponowany schemat oczekiwania (funkcja `aio_suspend`). Każde wywołanie powoduje konieczność przesyłania oraz skanowania całej tablicy rekordów, na które oczekujemy, co staje się nieefektywne przy dużej ich liczbie. Jest to powielenie tej samej wady, którą posiadają klasyczne funkcje `select` i `poll` [gammo]. Co gorsza, przy skanowaniu dla każdej z operacji musimy wywoływać funkcję `aio_error`. Należałoby dążyć do rozdzielenia rejestracji rekordów kontrolnych, którymi jesteśmy zainteresowani, od oczekiwania na nie [c10k, lemon].

## 3.3. Linux

Rekord kontrolny operacji asynchronicznej w systemie Linux zawiera wszystkie wymagane przez POSIX pola.

```
struct iocb {
    __u64 aio_data;        /* do wykorzystania przez użytkownika  */
    __u32 PADDED(aio_key, aio_reserved1);     /* identyfikator    */
    __u16 aio_lio_opcode;  /* kod przy zlecaniu wielu operacji    */
    __s16 aio_reqprio;     /* priorytet                           */
    __u32 aio_fildes;      /* deskryptor pliku                    */
    __u64 aio_buf;         /* bufor do odczytu-zapisu             */
    __u64 aio_nbytes;      /* liczba bajtów do odczytu-zapisu     */
    __s64 aio_offset;      /* pozycja w pliku,od której zaczynamy */
    __u64 aio_reserved2;   /* zarezerwowane, na wsk. sygnału      */
    __u64 aio_reserved3;   /* zarezerwowane                       */
};
```

Pola określające priorytet operacji `aio_reqprio` oraz sygnał, który ma być wysłany do procesu `aio_reserved2` w chwili obecnej istnieją tylko dla zachowania zgodności. Ich funkcjonalność nie jest jeszcze zaimplementowana. Polu `aio_data` możemy nadać dowolną wartość. Zostanie ona nam przekazana w rekordzie zdarzenia (np. można ją wykorzystać jako wskaźnik do rekordu prywatnego zawierającego dodatkowe informacje o operacji). Pole `aio_key` używane jest przez jądro do identyfikacji zlecenia.

Aby wykonywać operacje asynchroniczne w Linuksie należy najpierw utworzyć dla nich kontekst

```
int io_setup(int maxevents, aio_context_t *ctxp).
```

Każda operacja wykonuje się w dokładnie jednym kontekście. W tym samym kontekście może natomiast wykonywać się wiele operacji, odnoszących się do dowolnych deskryptorów. Parametr `maxevents` określa ile co najmniej zdarzeń zakończenia ma móc przechowywać tworzony kontekst. Bufor do przechowywania zdarzeń tworzony jest raz i ma stałą wielkość. Liczba zdarzeń ma wpływ na liczbę wykonujących się asynchronicznie operacji. Zlecenie nowej operacji powiedzie się tylko wtedy, gdy będzie zagwarantowane miejsce w buforze na zdarzenie jej zakończenia, w przeciwnym wypadku przy próbie zlecenia przekazany zostanie błąd [`EAGAIN`]. Po pomyślnym zakończeniu działania funkcji `io_setup` identyfikator nowo utworzonego kontekstu zostanie skopiowany pod adres `ctxp`.

By zlecić operację asynchroniczną, należy wywołać funkcję

```
int io_submit(aio_context_t ctx, long nr, struct iocb *iocbs[]).
```

Za jej pomocą zleca się wykonanie `nr` operacji. Wskaźniki do ich rekordów kontrolnych znajdują się w tablicy `iocbs`. Funkcja przekazuje liczbę operacji, które udało się zlecić (operacje z tablicy zlecane są kolejno). W przypadku, gdy nie uda się zlecić żadnej operacji przekazywany jest kod błędu dla pierwszej z nich.

Funkcja

```
int io_getevents(aio_context_t ctx, long min_nr, long nr,
                 struct io_event *events, struct timespec *timeout)
```

służy do oczekiwania na wykonanie się przynajmniej `min_nr` spośród operacji wykonujących się w kontekście `ctx`. W przypadku wykonania się przynajmniej minimalnej liczby `min_nr` operacji, na które oczekujemy lub po minięciu maksymalnego czasu oczekiwania `timeout` funkcja przekazuje liczbę pobranych zdarzeń zakończenia oraz wypełnia nimi tablicę `events` (każdej zakończonej operacji odpowiada dokładnie jedno zdarzenie). Rekord zdarzenia ma postać:

```
struct io_event {
    __u64 data;  /* wartość pola aio_data z rekordu kontrolnego */
    __u64 obj;   /* wskaźnik do rekordu kontrolnego struct iocb */
    __s64 res;   /* rezultat operacji                           */
    __s64 res2;  /* rezultat pomocniczy                         */
}
```

gdzie `obj` jest wskaźnikiem do rekordu kontrolnego zdarzenia, `data` wartością podaną przez użytkownika w momencie zlecania operacji (pole `aiocb->data`), a `res` wynikiem, jakim operacja się zakończyła. (W jądrze 2.6.1 pole `res2` nie jest używane).

Za pomocą funkcji

```
int io_cancel(aio_context_t ctx, struct iocb *iocb,
              struct io_event *event)
```

można próbować odwołać operację, podając wskaźnik do jej rekordu kontrolnego. (W jądrze 2.6.1 funkcja ta nie jest zaimplementowana dla żadnego systemu plików).

Do usuwania kontekstu służy funkcja

```
int io_destroy(aio_context_t ctx).
```

W momencie jej wykonania proces traci kontrolę nad kontekstem, a system operacyjny podejmuje próbę odwołania wszystkich związanych z kontekstem operacji, które jeszcze się nie zakończyły.

### 3.3.1. Przykład zastosowania

By lepiej pokazać działanie interfejsu zamieszczono tutaj prosty program w C, który z niego korzysta. Przykładowy program otwiera plik *test.dat*, zleca wykonanie dziesięciu operacji asynchronicznego odczytu oraz oczekuje na ich zakończenie.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <aio.h>

#define REQNO 10

int main()
```

```
{
  /* deskryptor, z którego będziemy czytać */
  int fd;

  /* kontekst operacji asynchronicznych */
  aio_context_t ioctx;

  /* tablica bloków kontrolnych operacji asynchornicznych */
  struct iocb *iocbs[REQNO];

  /* tablica z buforami, do których będą odczytywane dane */
  char *iobufs[REQNO];

  /* tablica zdarzeń zakończenia operacji asynchronicznych */
  struct io_event ioevs[REQNO];

  /* czas, co jaki będą pobierane zdarzenia zakończenia */
  struct timespec waitinterval = { tv_sec: 1, tv_nsec: 0 };

  /* zmienne pomocnicze */
  int i, res, left;

  /* otwieramy plik, z którego odczytywane będą dane */
  fd = open("test.dat", O_RDONLY | O_DIRECT);
  if (fd == -1) {
    perror("open");
    exit(-1);
  }

  /* tworzymy nowy kontekst dla operacji asynchronicznych */
  ioctx = 0;
  res = io_setup(REQNO, &ioctx);
  if (res < 0) {
    fprintf(stderr, strerror(-res));
    exit(-1);
  }

  /* tworzymy rekordy kontrolne operacji asynchronicznych */
  for (i = 0; i < REQNO; i++) {

    /* alokujemy pamięć na rekord kontrolny */
    if ((iocbs[i] = malloc(sizeof (struct iocb))) == NULL) {
      perror("malloc iocb");
      exit(-1);
    }

    /* alokujemy pamięć na bufor, do którego odczytane będą dane */
    if ((iobufs[i] = valloc(4096)) == NULL) {
      perror("valloc iobufs");
      exit(-1);
    }

    /* inicjujemy rekord kontrolny */
    memset(iocbs[i], 0, sizeof(struct iocb));
    iocbs[i]->aio_data = i;
```

```
    iocbs[i]->aio_fildes = fd;
    iocbs[i]->aio_lio_opcode = IO_CMD_PREAD;
    iocbs[i]->aio_reqprio = 0;
    iocbs[i]->aio_buf = iobufs[i];
    iocbs[i]->aio_nbytes = 4096;
    iocbs[i]->aio_offset = 4096 * i;
  }

  /* zlecamy wykonanie przygotowanych operacji */
  res = io_submit(ioctx, REQNO, iocbs);
  if (res < 0) {
   fprintf(stderr, strerror(-res));
   exit(-1);
  }

  /* oczekujemy na wykonanie się wszystkich zadań */
  left = REQNO;
  while (left > 0) {

    /* pobieramy zdarzenia zakończenia */
    res = io_getevents(ioctx, 1, REQNO, ioevs, &waitinterval);

    /* jeżeli jakieś zadania się zakończyły */
    if (res > 0) {

      /* zmniejszamy liczbę zadań, na które czekamy */
      left -= res;

      /* wypisujemy na ekran informacje o zakończonych zdarzeniach */
      for (i = 0; i < res; i++) {
        printf("%d: data: %ld, res: %ld\n",
               i, ioevs[i].data, ioevs[i].res);
      }
    }
  }

  /* usuwamy kontekst */
  if (io_destroy(ioctx) != 0) {
    fprintf(stderr, "io_destroy failed!");
    exit(-1);
  }

  return 0;
}
```

### 3.3.2. Wady i zalety

Interfejs Linuksowy jest wydajniejszy od Posiksowego. Do obsługi operacji asynchronicznej wystarczają dwa wywołania systemowe: do zlecenia (`io_submit`) i do otrzymania informacji zwrotnej (`io_getevents`). Dzięki oddzieleniu faz rejestracji zdarzeń, którymi jesteśmy zainteresowani od oczekiwania na nie możliwe jest współbieżne oczekiwanie na tym samym kontekście przez wiele wątków tego samego procesu. Można również oczekiwać na wiele operacji bez konieczności przesyłania i późniejszego skanowania całej tablicy rekordów kontrolnych przy każdym wywołaniu.

Niewątpliwą wadą jest brak możliwości oczekiwania na zdarzenia zakończenia operacji asynchronicznej w połączeniu ze zdarzeniami innych typów (np. gotowości deskryptorów do odczytu). Proponowana w niniejszej pracy modyfikacja jądra systemu Linux ma na celu dodanie takiej właśnie funkcjonalności.

Niestety w jądrze 2.6.1 operacje asynchroniczne zaimplementowane są jedynie dla kilku systemów plików (EXT2, EXT3, NFS, XFS, JFS) i tylko dla plików otwartych z wyłączonym buforowaniem (z flagą `O_DIRECT`).

## 3.4. Windows

W systemach Windows do odczytu synchronicznego i asynchroniczego służy ta sama funkcja systemowa

```
BOOL ReadFile(HANDLE hFile,
              LPVOID lpBuffer,
              DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead,
              LPOVERLAPPED lpOverlapped).
```

By dane zostały odczytane asynchronicznie plik (do którego uchwytem jest `hFile`) musi być otwarty z flagą `FILE_FLAG_OVERLAPPED` oraz należy podać wskaźnik do rekordu kontrolnego `OVERLAPPED`, w którym znajdować się będą dodatkowe parametry, wymagane przez operację asynchroniczną. W przypadku, gdy powiedzie się zlecenie operacji asynchronicznej (lub odczyt synchroniczny) funkcja przekazuje `true`; w przypadku niepowodzenia — `false` oraz ustawia odpowiedni kod błędu. Zwróćmy uwagę na to, że liczba przeczytanych bajtów zostanie wpisana pod wskazywany przez parametr `lpNumberOfBytesRead` adres w pamięci (wskaźnik na `DWORD`).

Rekord kontrolny operacji AIO ma postać

```
typedef struct _OVERLAPPED {
        ULONG_PTR Internal;
        ULONG_PTR InternalHigh;
        DWORD Offset;
        DWORD OffsetHigh;
        HANDLE hEvent;
} OVERLAPPED.
```

`Offset` jest przesunięciem w pliku, od którego ma się rozpocząć operacja; `hEvent` zdarzeniem, które zostanie wysłane do procesu w momencie jej zakończenia. Systemy operacyjne Windows umożliwiają procesowi tworzenie zdarzeń (funkcja `CreateEvent`) i późniejsze oczekiwanie na ich zajście za pomocą jednej spośród funkcji oczekiwania (ang. *wait functions*). Szczegóły dotyczące obsługi zdarzeń można znaleźć w [msdn].

W Windows istnieje również wyspecjalizowana funkcja do odczytu asynchronicznego

```
BOOL ReadFileEx(HANDLE hFile,
                LPVOID lpBuffer,
                DWORD nNumberOfBytesToRead,
                LPOVERLAPPED lpOverlapped,
                LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine).
```

Przyjmuje ona dodatkowo parametr `lpCompletionRoutine` — wskaźnik na procedurę zwrotną dla operacji asynchronicznych. Procedura ta wykonywana jest w momencie gdy (zachodzą oba warunki):

1. zakończy się operacja asynchroniczna,
2. zlecający ją wątek oczekuje na zdarzenia za pomocą jednej z funkcji: `SleepEx`, `MsgWaitForMultipleObjectsEx`, `WaitForSingleObjectEx` lub `WaitForMultipleObjectsEx` z ustawioną flagą `fAlertable`.

Procedura zwrotna dla AIO jest przypadkiem szczególnym mechanizmu asynchronicznych procedur zwrotnych (ang. *Asynchronous Procedure Call*); szczegółowe informacje na jego temat znajdują się w [msdn].

Zlecenie asynchronicznego zapisu realizowane jest przez analogiczne funkcje `WriteFile` oraz `WriteFileEx`.

Do pobrania informacji zwrotnej o tym jak zakończyła się pojedyncza operacja asynchroniczna oraz do ewentualnego oczekiwania na jej zakończenie służy funkcja

```
BOOL GetOverlappedResult(HANDLE hFile,
                         LPOVERLAPPED lpOverlapped,
                         LPDWORD lpNumberOfBytesTransferred,
                         BOOL bWait).
```

Jej argumentami są uchwyt do pliku `hFile`, na którym wykonywana była operacja AIO; wskaźnik do rekordu `OVERLAPPED` opisującego operacje; flaga `bWait`, określająca czy chcemy oczekiwać na zakończenie, czy dokonać nieblokującego sprawdzenia oraz wskaźnik `lpNumberOfBytesTransferred` do adresu w pamięci, pod którym wpisana zostanie liczba bajtów jakie udało się odczytać (zapisać).

Kolejnym mechanizmem, umożliwiającym oczekiwanie na zakończenie operacji AIO jest port zakończenia wejścia-wyjścia (ang. *IO Completion Port*). Nowy port tworzy się za pomocą funkcji `CreateIoCompletionPort`. Ta sama funkcja służy również do zarejestrowania w *IOCP* uchwytu pliku. Po zarejestrowaniu zdarzenia informujące o wykonanych operacjach asynchronicznych na uchwycie wysyłane będą do wskazanego portu. Port jest faktycznie pewnego rodzaju kolejką zdarzeń. Wątek za pomocą funkcji `GetQueuedCompletionStatus` może oczekiwać na porcie na zdarzenia. Do portu można również wysyłać zdarzenia; służy do tego funkcja `PostQueuedCompletionStatus`. Na porcie może oczekiwać wiele wątków tego samego procesu, jest to dodatkowo wspierane przez możliwość ustalenia stopnia współbieżności portu — liczby określającej ile maksymalnie wątków, spośród oczekujących, będzie się wykonywać równocześnie.

### 3.4.1. Wady i zalety

W systemie Windows interfejs do obsługi AIO jest bardzo rozbudowany. Dostarcza wiele mechanizmów do oczekiwania, często o zbliżonej lub zachodzącej na siebie funkcjonalności. Wątek, by dowiedzieć się o zakończeniu operacji AIO, może:

- oczekiwać bezpośrednio na zakończenie operacji (`GetOverlappedResult`),
- oczekiwać na nadejście wskazanego zdarzenia (*wait functions*),
- oczekiwać na wywołanie procedury asynchronicznej (`SleepEx`, `MsgWaitForMultipleObjectsEx`, `WaitForSingleObjectEx`, `WaitForMultipleObjectsEx`),

- oczekiwać na zdarzenie na porcie *IOCP* (`GetQueuedCompletionStatus`).

Trudno jest ocenić, które rozwiązania są najwłaściwsze dla jakich zastosować oraz w jakim stopniu przekładają się one na rzeczywiste mechanizmy jądra systemu operacyjnego, a w jakim są implementacjami z poziomu bibliotek.

## 3.5. FreeBSD

System FreeBSD udostępnia Posiksowy interfejs do obsługi operacji asynchronicznych (dokładnie taki jak opisany w podrozdziale 3.2). Z powodu niedogodności, jakie wiążą się z używaniem go do otrzymywania informacji zwrotnej (czy i jak zakończyła się operacja AIO), umożliwia użycie do tego celu mechanizmu uogólnionych kolejek zdarzeń (ang. *kqueue*) [lemon].

Proces, pragnący korzystać z tego mechanizmu musi najpierw utworzyć kolejkę za pomocą funkcji

```
int kqueue(void).
```

W wyniku jej działania przekazywany jest reprezentujący ją deskryptor pliku.

Funkcja

```
int kevent(int kq, int nchanges, struct kevent **changelist,
           int nevents, struct kevent *eventlist,
           struct timespec *timeout)
```

służy zarówno do ustalania, którymi informacjami i o jakich zasobach jesteśmy zainteresowani, jak i do oczekiwania na nie. Parametr `kq` określa deskryptor kolejki. Zdarzenie opisywane jest przez rekord:

```
struct kevent {
    uintptr_t ident;        /* identyfikator zasobu            */
    short     filter;       /* filtr                           */
    u_short   flags;        /* flagi akcji                     */
    u_int     fflags;       /* flagi dla filtru                */
    intptr_t  data;         /* dane dla filtru                 */
    void      *udata;       /* wskaźnik do danych użytkownika  */
}.
```

Gdy chcemy zmodyfikować zasoby, które są zarejestrowane w kolejce `kq` podajemy tablicę modyfikacji `changelist` rozmiaru `nevents`. Każda modyfikacja (opisana przez rekord `struct kevent`) ma flagę akcji `flags`, określającą na czym wprowadzana zmiana ma polegać (np. `EV_ADD` dla dodania nowego typu zdarzeń, `EV_DELETE` dla usunięcia). Pole `filtr` określa typ zdarzenia, może to być:

- `EVFILT_READ` — gotowość deskryptora do odczytu,
- `EVFILT_WRITE` — gotowość deskryptora do zapisu,
- `EVFILT_AIO` — monitorowanie zdarzeń zakończenia operacji AIO dotyczących deskryptora,
- `EVFILT_VNODE` — zdarzenia na v-węźle (usunięcie, zmiana atrybutów itp...),
- `EVFILT_PROC` — zdarzenia na procesie (zakończenie działania, wywołanie funkcji fork),

- `EVFILT_SIGNAL` — nadejście sygnału.

Pole `ident` określa zasób, którego dotyczy zdarzenie; w zależności od filtru może to być numer deskryptora, identyfikator procesu (pid), wartość sygnału itp. Za pomocą parametrów `fflag` i `data` możemy dokładniej określić, jakimi zdarzeniami jesteśmy zainteresowani.

By oczekiwać na zajście zarejestrowanych zdarzeń, musimy podać tablicę `eventlist`, do której zostaną one zapisane, oraz jej rozmiar `nevents`. Możemy również podać czas `timeout`, po którym oczekiwanie ma zostać przerwane.

### 3.5.1. Wady i zalety

Interfejs udostępniany przez FreeBSD przedstawia się najkorzystniej spośród prezentowanych tu rozwiązań. Dzięki kolejkom zdarzeń operacja asynchroniczna może być obsłużona za pomocą dwóch wywołań systemowych. Mechanizm powiadamiania jest bardzo ogólny. Rozdzielone są fazy rejestracji zdarzeń od oczekiwania na nie, co czyni go wydajnym oraz umożliwia współbieżne oczekiwanie wielu wątków na te same zdarzenia.

# Rozdział 4

# Mechanizmy jądra Linuksa

## 4.1. Wstęp

W rozdziale tym omówiona jest pokrótce napisana w ramach pracy łata dla jądra Linuksa oraz opisane jest dokładnie działanie mechanizmów jądra, w których wprowadza ona modyfikacje i z którymi współpracuje (AIO i *epoll*).

## 4.2. Łata

W ramach niniejszej pracy zostały opracowane modyfikacje jądra systemu Linux (w wersji 2.6.1), dzięki którym można będzie oczekiwać wspólnie na zdarzenia zakończenia operacji asynchronicznych wraz ze zdarzeniami innych typów. W tym celu została napisana łata, która włącza operacje asynchroniczne do mechanizmu powiadamiania *event-poll*.

Każdy kontekst operacji asynchronicznych można będzie powiązać z plikiem nowoutworzonego systemu plików *aiofs*, a plik systemu *aiofs* zarejestrować w *event-poll*. Opracowane zostały dwie wersje łaty: *zachowawcza*, ograniczająca się do dodania nowej funkcjonalności z zachowaniem pełnej zgodności jądra wstecz oraz *odważna* usuwająca z jądra mechanizmy, które w świetle zmian stały się redundantne i mniej efektywne.

### 4.2.1. Wersja *zachowawcza*

Do jądra została dodana nowa funkcja systemowa: `sys_io_bind(ioctx)`, która jako argument dostaje kontekst AIO i tworzy dla niego obiekt pliku, reprezentowany przez deskryptor (lub przekazuje już istniejący jeżeli kontekst jest już z jakimś powiązany). Możliwa jest rejestracja tak utworzonego deskryptora w *event-poll* i oczekiwanie, za pomocą funkcji `epoll_wait`, aż będzie on gotowy do odczytu (tzn. na odpowiadającym mu kontekście będzie oczekiwać zdarzenie zakończenia) lub zapisu (tzn. odpowiadający mu kontekst będzie miał miejsce na przyjęcie kolejnej operacji asynchronicznej).

### 4.2.2. Wersja *odważna*

Obecnie w Linuksie konteksty AIO opisywane są przez rekordy z przestrzeni jądra (`struct kioctx`), a lista wskaźników do wszystkich stworzonych przez proces kontekstów trzymana jest w polu `ioctx_list` rekordu `mm_struct` (por. p. 4.3.1). Przy tworzeniu nowego kontekstu użytkownik dostaje do niego identyfikator, na podstawie którego przy każdym odwołaniu wyszukuje się go na tej liście. W *odważnej* wersji w momencie tworzenia kontekstu tworzy się i przekazuje deskryptor pliku (w którym przechowywany jest wskaźnik do rekordu `struct`

kioctx), poprzez który można się potem do niego odwoływać. Lista kontekstów z rekordu mm_struct została usunięta.

Należy zwrócić uwagę, że obecnie przy odwołaniu się do kontekstu musi on zostać wyszukany z listy, co zajmuje czas liniowy — $O(n)$ (gdzie $n$ — liczba kontekstów otwartych przez proces). Dzięki odwoływaniu się poprzez tablicę deskryptorów unikamy tego problemu (dostęp jest realizowane w czasie stałym).

## 4.3. Działanie AIO

Podrozdział ten opisuje jak dokładnie zrealizowane są operacje asynchroniczne w jądrze systemu Linux 2.6.1. Ich implementacja znajduje się w pliku *fs/aio.c*.

### 4.3.1. Kontekst

Po stronie jądra kontekst dla operacji asynchronicznych opisywany jest przez rekord:

```
struct kioctx {
    atomic_t            users;      /* licznik użycia           */
    int                 dead;
            /* flaga, czy kontekst jest przeznaczony do usunięcia  */
    struct mm_struct    *mm;
            /* rekord mm_struct procesu, który utworzył kontekst   */
    unsigned long       user_id;
            /* identyfikator kontekstu posiadany przez użytkownika */
    struct kioctx       *next;
            /* następny kontekst na liście w mm_struct             */
    wait_queue_head_t   wait;
            /* kolejka do oczekiwania na zakończenie operacji      */
    spinlock_t          ctx_lock;
            /* blokada dla sekcji krytycznej kontekstu             */
    int                 reqs_active;
            /* liczba aktualnie wykonywanych operacji              */
    struct list_head    active_reqs;
            /* lista aktualnie wykonywanych operacji               */
    struct list_head    run_list;
            /* na wywłaszczone operacje - obecnie nieużywane       */
    unsigned            max_reqs;
            /* maksymalna liczba operacji na kontekście            */
    struct aio_ring_info ring_info;
            /* powiązane z kontekstem zdarzenia                    */
    struct work_struct  wq;
            /* kolejka do oczekiwania na zdarzenia                 */
}.
```

Lista wszystkich utworzonych przez proces kontekstów przechowywana jest w polu ioctx_list odpowiadającego procesowi rekordu mm_struct (por. rys. 4.1). Konteksty nie są dziedziczone przez proces potomny.

Po utworzeniu nowego kontekstu za pomocą funkcji sys_io_setup użytkownik, do jego identyfikacji, otrzymuje zmienną typu aio_context_t (zdefiniowanego w pliku *include/linux/*

*aio_abi.h* jako `unsigned long`). Ma ona wartość równą wartości pola `user_id` odpowiadającego jej rekordu jądra `kioctx`.

Przy wywołaniu funkcji systemowych, w których podawany jest identyfikator kontekstu (typu `aio_context_t`), jądro, za pomocą funkcji `lookup_ioctx`, przegląda listę wszystkich kontekstów i porównuje podaną przez użytkownika wartość z polem `user_id` rekordu. Gdy uda się znaleźć rekord o podanym identyfikatorze zwiększany jest o jeden jego licznik użycia i przekazywany jest wskaźnik do niego. Po skończeniu używania kontekstu jest on zwalniany za pomocą funkcji `put_ioctx`, która zmniejszy jego wskaźnik użycia o jeden.

Informacje o zdarzeniach zakończenia przechowywane są w rekordach:

```
struct aio_ring_info {
    unsigned long mmap_base;
    unsigned long mmap_size;
    struct page   **ring_pages;
                  /* bufor cykliczny do przechowywania zdarzeń  */
    spinlock_t    ring_lock; /* blokada dla sekcji kryt. bufora */
    long          nr_pages;  /* liczba stron pamięci w buforze  */
    unsigned      nr;
                  /* rozmiar bufora cyklicznego(liczba zdarzeń) */
    unsigned      tail;      /* indeks ostatniego zdarzenia     */
    struct page   internal_pages[AIO_RING_PAGES];
                  /* do szybkiej inicjacji pola ring_pages      */
}
```

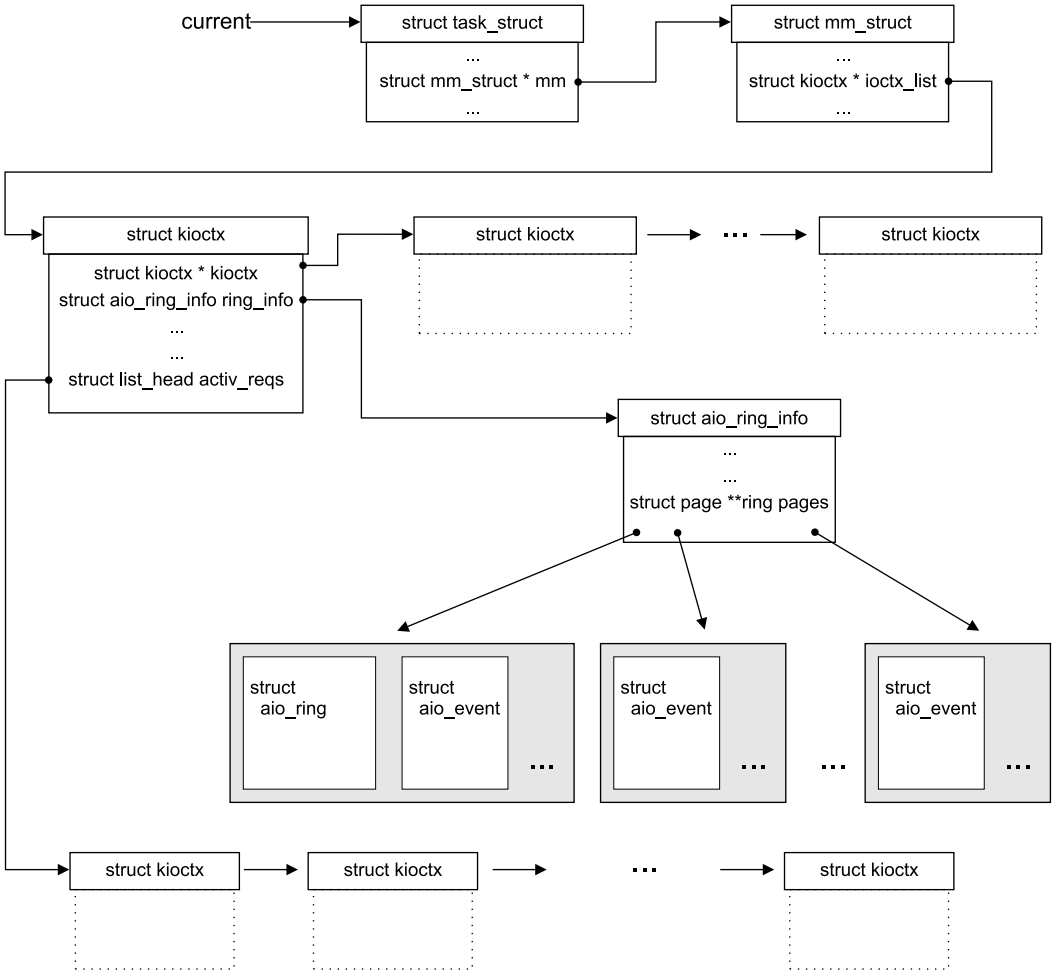
oraz:

```
struct aio_ring {
    unsigned        id;      /* identyfikator na potrzeby jądra */
    unsigned        nr;      /* rozmiar bufora cyklicznego      */
    unsigned        head;    /* indeks pierwszego zdarzenia     */
    unsigned        tail;    /* indeks ostatniego zdarzenia     */
    unsigned        magic;
    unsigned        compat_features;
    unsigned        incompat_features;
    unsigned        header_length;  /* rozmiar rekordu aio_ring */
    struct io_event io_events[0];
}
```

Wskaźnik do pierwszego z nich (`aio_ring_info`) trzymany jest w polu `ring_info kioctx`. Rekord ten zawiera wskaźnik `ring_pages` do tablicy, w której trzymane są adresy stron pamięci przeznaczonych na cykliczny bufor zdarzeń. Na początku bufora (na początku strony o indeksie 0) zapisany jest rekord `aio_ring` zawierający informacje pomocnicze do jego obsługi (por. rys. 4.1). Pole `internal_pages` rekordu `aio_ring_info` służy do domyślnej inicjacji tablicy `ring_pages` tablicą wskaźników rozmiaru `AIO_RING_PAGES`, jeżeli wymagana przez użytkownika liczba zdarzeń, jaką kontekst powinien móc jednocześnie obsługiwać, zmieści się na tylu stronach pamięci. W przeciwnym wypadku tablica ta alokowana jest dynamicznie.

### 4.3.2. Synchronizacja dostępu

Obsługę sekcji krytycznej dla listy kontekstów umożliwia pole `ioctx_list_lock` typu `rwloct_t` rekordu `mm_struct`. Jest to blokada wirująca, implementująca problem czytelników i pisarzy.




Rysunek 4.1: Kontekst AIO

Wątek jądra pragnący przeglądać listę musi przejść przez blokadę jako czytelnik, wątek, który chce usunąć lub dodać do listy kontekst — jako pisarz.

Zanim operacja asynchroniczna zostanie zlecona, zwiększany jest licznik użycia `users` rekordu kontekstu. Zmniejszenie licznika następuje w momencie zakończenia operacji. Funkcja systemowa `io_destroy`, służąca do zamykania kontekstu, usuwa jedynie kontekst z listy w `mm_struct` i ustawia wartość flagi `dead` na jeden. Pamięć jest zwalniana w momencie zmniejszania licznika użycia, gdy spadnie on do zera i flaga `dead` jest zaznaczona. Dzięki temu kontekst nie zostaje usunięty z pamięci, aż nie dobiegną końca wszystkie wykonujące się na nim operacje.

Każdy kontekst zabezpiecza własną sekcje krytyczną poprzez blokadę wirującą `ctx_lock`. Wewnątrz sekcji krytycznej może znajdować się tylko jeden wątek jądra, bez względu na to, czy ma zamiar odczytywać czy zapisywać. Z rekordu kontekstu oraz powiązanej z nim tablicy zdarzeń ma również miejsce odczytywanie danych bez blokowania sekcji krytycznej, ale tylko wtedy gdy:

- został zwiększony licznik użycia (pamięć kontekstu nie może zostać zwolniona),
- odczyt jest wykonywany za pomocą operacji atomowej (by nie było konfliktu z ewentualnym zapisem).

Do tego celu używane jest atomowe mapowanie stron pamięci (funkcje `kmap_atomic` i `kmap`).

Dodatkowo, w sekcji krytycznej rekordu `aio_ring_info` znajduje się blokada wirująca `ring_lock`, która używana jest przy odczytywaniu zdarzeń z bufora. (Użycie blokady spowodowane jest koniecznością modyfikowania indeksów przy pobraniu zdarzeń).

### 4.3.3. Operacje plikowe

Operacje asynchroniczne na plikach, tak jak operacje synchroniczne (odczyt, zapis) są w dużym stopniu zależne od typu pliku, którego dotyczą. Dlatego też w jądrach 2.6, w porównaniu z jądrami 2.4 [linkern], pojawiły się nowe metody w rekordzie `file_operations`. Są to: `aio_read` — służąca do asynchronicznego odczytu, `aio_write` — do zapisu, `aio_fsync` — do synchronizacji pliku (na razie niezaimplementowana dla żadnego systemu plików). Dzięki temu można zaimplementować AIO dla nowotworzonego systemu plików.

### 4.3.4. Działanie udostępnianych funkcji systemowych

#### A. io_setup

Funkcji systemowej `io_setup` odpowiada funkcja

```
long sys_io_setup(unsigned nr_events, aio_context_t *ctxp)
```

z przestrzeni jądra. Tworzy ona nowy kontekst dla operacji asynchronicznych:

- Sprawdzana jest poprawność argumentów.
- Jeżeli argumenty są poprawne, to wywoływana jest funkcja `ioctx_alloc(nr_events)`.
- Kopiowany jest do przestrzeni użytkownika odpowiedni identyfikator kontekstu.

Funkcja `ioctx_alloc` alokuje pamięć i inicjuje nowy kontekst:

- Z alokatora płytowego pobierana jest pamięć dla rekordu `struct kioctx`.
- Jeżeli na pomieszczenie podanej przez użytkownika liczby zdarzeń wystarcza `AIO_RING_PAGES` stron pamięci, to tablica `ring_pages` inicjowana jest wartością pola `internal_pages`. W przeciwnym przypadku tablica `ring_pages` alokowana jest dynamicznie.
- Rekord kontekstu jest inicjowany oraz wstawiany do listy kontekstów w `mm_struct`.

#### B. io_submit

Zlecenie przez użytkownika operacji AIO poprzez io_submit powoduje wywołanie funkcji

```
long sys_io_submit(aio_context_t ctx_id, long nr, struct iocb
                   __user **iocbpp),
```

wewnątrz której:

- Za pomocą funkcji `lookup_ioctx` z listy wyszukiwany jest podany kontekst i zwiększany jego licznik użycia.
- Dla każdej z `nr` zlecanych operacji wywoływana jest funkcja `io_submit_one` dopóty, dopóki zlecenia kończą się powodzeniem.

- Poprzez `put_ioctx` zmniejszany jest licznik użycia kontekstu.
- Jeśli udało się zlecić przynajmniej jedną operację, to przekazywana jest liczba operacji jakie udało się zlecić, w przeciwnym przypadku — kod błędu pierwszej operacji.

Funkcja

```
io_submit_one(struct kioctx *ctx, struct iocb __user
                *user_iocb, struct iocb *iocb)
```

zleca wykonanie pojedynczej operacji AIO:

- Sprawdzana jest poprawność argumentów.
- Za pomocą funkcji `fget` na podstawie deskryptora pliku, którego ma dotyczyć operacja uzyskiwany jest wskaźnik do rekordu `struct file`.
- Poprzez funkcje `aio_get_req` alokowany i inicjowany jest blok kontrolny operacji asynchronicznej z poziomu jądra `struct kiocb`.
- W zależności od tego, jaką operacje chcemy zlecić, wywoływana jest odpowiednia metoda pliku, którego ma ona dotyczyć: `aio_read`, `aio_write` lub `aio_fsync`.

### C. io_getevents

Odpowiednikiem funkcji `io_getevents` jest

```
sys_io_getevents(aio_context_t ctx_id, long min_nr, long nr,
                 struct io_event *events, struct timespec *timeout)
```

wewnątrz której:

- Za pomocą funkcji `lookup_ioctx` z listy wyszukiwany jest podany kontekst i zwiększany jego licznik użycia.
- Wywoływana jest funkcja `read_events`.
- Poprzez `put_ioctx` kontekst jest odkładany i licznik jego użycia zmniejszany.

Funkcja

```
int read_events(struct kioctx *ctx, long min_nr, long nr,
                struct io_event *event, struct timespec *timeout):
```

- Za pomocą `aio_read_event` odczytuje z bufora zdarzeń `aio_ring_info` zdarzenie po zdarzeniu, aż osiągnie ich maksymalną liczbę `nr` lub aż zdarzenia w buforze się skończą.
- Jeżeli odczytana liczba zdarzeń jest mniejsza niż `min_nr`, to zasypia, na czas nie dłuższy niż `timeout`, na kolejce AIO kontekstu `ctx->wait` w oczekiwaniu na kolejne.

**D. io_canc**

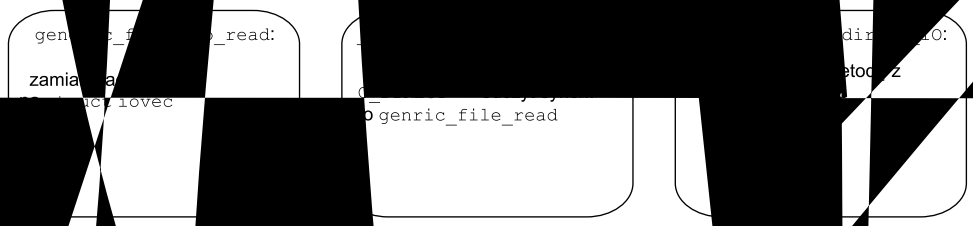
Próba anul a opera ynchron wołanie systemowe

```
ng sys_io cancel(aio ont b,
                 str ct i
```

Sprowa si d ywołania metody acji as niczne ct tóra ma zostać odw dne, p opera anulow że mieć różną pos racji obec j chwili m ancel nie jest zai ów p

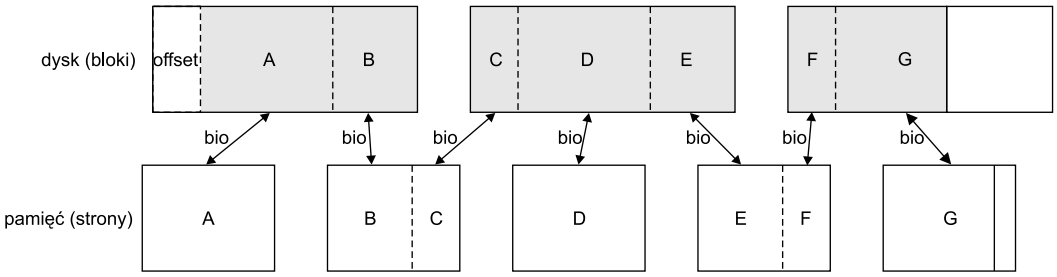
### 4.3.5. Przykła czytu d a EXT2

punkcie tym o na jest ealizacja o ku z EXT2. Jest ona im entow poprzez m _op wywoły aną przez cję i ubmit_one



zamienia operację opisaną przez rekord iocb na operację zlecenie bezpośredniego odczytu (`direct-io`), składające się z zestawu zleceń `bio`. Każde zlecenie `bio` jest tak skonstruowane, żeby możliwe było szybkie bezpośrednie skopiowanie danych z dysku pod adres pamięci, dlatego też pojedyncze zlecenie obejmuje skopiowanie fragmentu jednego bloku dyskowego do fragmentu jednej strony pamięci (por. rys. 4.4). Dokładniejszy opis tego mechanizmu można znaleźć w [bio].

- Dla operacji tworzony jest rekord `struct dio`.
- Rekord zlecenia dio jest inicjowany za pomocą funkcji `do_direct_IO`, która tworzy dla niego zestaw pojedynczych operacji bio.
- Za pomocą wywołania `dio_bio_submit` zestaw zleceń wstawiany jest do kolejki urządzenia blokowego (*driver/block/ll_rw_blk.c*).
- Jeżeli operacja ma być wykonana synchronicznie, to wywoływana jest funkcja `dio_complete` czekająca na jej zakończenie.
- Jeżeli operacja ma wykonywać się asynchronicznie, to właśnie zlecenie jej wykonania się zakończyło, *odkorkowuje się* więc kolejki urządzeń blokowych za pomocą `blk_run_queues`.




Rysunek 4.4: Zlecenie Direct IO

Dla operacji asynchronicznej informacja o jej zakończeniu przekazywana jest poprzez funkcję

```
finished_one_bio(struct dio *dio)
```

wykonującą się przy zakończeniu każdej z pojedynczych operacji `bio` należących do `dio`. Jeżeli kończąca się operacja `bio` była ostatnią (licznik `dio->bio_count` spadnie do zera) i `dio` było operacją asynchroniczną, to wywoływana jest funkcja `aio_complete` z rezultatem, jakim operacja `dio` się zakończyła.

Funkcja

```
aio_complete(struct kiocb *iocb, long res, long res2)
```

zdefiniowana w pliku *fs/aio.c*:

- wstawia zdarzenie zakończenia do bufora zdarzeń `aio_ring`,
- budzi procesy czekające na zdarzenia w kolejce `ctx->wait`,

co kończy obsługę operacji asynchronicznej.

## 4.4. Działanie *event-poll*

W niniejszym podrozdziale opisano mechanizm *event-poll* — linuksowego następcę funkcji systemowej `poll`. Podany jest jego interfejs, realizacja na poziomie jądra oraz sposób, w jaki można dodać do niego obsługę nowego systemu plików.

### 4.4.1. Interfejs

W systemach Unix klasyczna funkcja `poll` ma zazwyczaj interfejs postaci [vah]:

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout)
```

gdzie `ufds` jest tablicą rozmiaru `nfds`, rekordów

```
struct pollfd {
    int fd;       /* deskryptor, którym jesteśmy zainteresowani */
    short events;
       /* zdarzenia gotowości, którymi jesteśmy zainteresowani  */
    short revents; /* przekazywane zdarzenia gotowości          */
}
```

zawierających numery deskryptorów `fd` oraz zdarzenia `events` (określające gotowość deskryptora do odczytu lub zapisu), którymi jesteśmy zainteresowani. Po zakończeniu działania funkcja przekazuje liczbę deskryptorów, na których zaszły oczekiwane zdarzenia oraz wypełnia nimi odpowiednie pola `revents` w tablicy `ufds`.

W rozdziale 1 zwracaliśmy już uwagę na wady tego mechanizmu.

- Konieczność wielokrotnego skanowania całej tablicy deskryptorów:
  - przez SO przy każdym wywołaniu funkcji,
  - przez zleceniodawcę po każdym powrocie z funkcji,

  powodująca koszt liniowy względem liczby deskryptorów.
- Niemożność oczekiwania na gotowość deskryptorów z tego samego zestawu przez kilka wątków jednego procesu.

Z ich przyczyny powstał w systemie Linux nowy mechanizm powiadamiania zdarzeniowego (ang. *event-poll*, w skrócie *epoll*).

By korzystać z *epoll* trzeba najpierw za pomocą funkcji:

```
int epoll_create(int size)
```

utworzyć deskryptor *epoll*, w którym rejestrowane będą deskryptory oraz na którym będzie można oczekiwać na ich gotowość. *Event-poll* reprezentowany jest przez liczbę naturalną będącą deskryptorem pliku. W momencie tworzenia podawany jest rozmiar `size`, który określa maksymalną liczbę deskryptorów, jakie chcielibyśmy w nim zarejestrować (rozmiar ten nie jest traktowany tak ściśle, jak w przypadku kontekstu AIO, stanowi on jedynie wskazówkę dla jądra).

Za pomocą funkcji

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event),
```

w zależności od argumentu `op`, możemy dodać nowy lub usunąć stary deskryptor z deskryptora *epoll* `epfd` oraz ustalić dla niego zdarzenia, którymi jesteśmy zainteresowani.

Do oczekiwania na gotowość zarejestrowanych deskryptorów służy funkcja

```
int epoll_wait(int epfd, struct epoll_event * events,
               int maxevents, int timeout).
```

Podawany jest deskryptor *epoll*, na którym będziemy oczekiwać, tablica na zdarzenia gotowości rozmiaru `maxevents` oraz maksymalny czas oczekiwania `timeout`.

Zdarzenie określające gotowość deskryptorów opisywane jest przez rekord

```
struct epoll_event {
        __uint32_t   events; /* zdarzenia,  które zaszły        */
        epoll_data_t data;   /* wskaźnik do danych użytkownika  */
};
typedef union epoll_data {
        void         *ptr;
        int          fd;
        __uint32_t   u32;
        __uint64_t   u64;
} epoll_data_t;
```

gdzie `events` to mapa bitowa zdarzeń, które zaszły (lub, przy rejestracji, którymi jesteśmy zainteresowani), `data` — dowolny wskaźnik, podawany przez użytkownika w tracie rejestracji i przekazywany mu przy zajściu zdarzenia gotowości (może on wskazywać na deskryptor pliku `fd` lub cokolwiek innego).

Jak już wcześniej wspomniano, deskryptor *epoll* jest deskryptorem pliku. Dzięki temu możemy zarejestrować deskryptor *epoll* A w innym deskryptorze epoll B i oczekiwać na B, aż A będzie gotowy do odczytu (czyli aż będziemy mogli odczytać z A zdarzenie gotowości jednego z zarejestrowanych na nim deskryptorów). Celem napisanej w obrębie niniejszej pracy łaty jest umożliwienie takiego właśnie rejestrowania kontekstów AIO w deskryptorach *epoll*.

### 4.4.2. System plików *eventpollfs*

By umożliwić odwoływanie się do *epoll* poprzez deskryptor w systemie operacyjnym tworzony jest specjalny system plików.

System plików *eventpollfs* rejestrowany jest i montowany wewnątrz funkcji `eventpoll_init` (*fs/eventpoll/c*) wywoływanej w momencie ładowania modułu `eventpoll`. Rejestracja odbywa się poprzez wywołanie funkcji `register_filesystem` dla zmiennej `eventpoll_fs_type` opisującej typ. Następnie za pomocą funkcji `kern_mount` montowana jest jego jedna instancja (`eventpoll_mnt`), do której należeć będą wszystkie tworzone przez `epoll_create` pliki.

Dla systemu plików typu *epoll* określona jest nazwa i dwie operacje:

```
static struct file_system_type eventpoll_fs_type = {
        .name       = "eventpollfs",
        .get_sb     = eventpollfs_get_sb,
        .kill_sb    = kill_anon_super,
};
```

Do tworzenia i usuwania superbloku używane są standardowe funkcje dla pseudo-systemów plików (nie związanych z żadnym konkretnym urządzeniem) — `get_sb_pseudo` (`eventpollfs_get_sb` jest na nią nakładką) oraz `kill_anon_super`.

Udostępniana użytkownikowi funkcja `epoll_create` służąca do tworzenia nowych plików *epoll* wywołuje funkcję jądra

```
long sys_epoll_create(int size).
```

Wewnątrz niej wywoływane są dwie funkcje:

- `ep_getfd(&fd, &inode, &file)` — tworząca obiekty pliku i i-węzeł systemu plików *epollfs*,
- `ep_file_init(file, hashbit)` — inicjująca plik i przekazująca numer deskryptora.

Funkcja jądra

```
static int ep_getfd(int *efd, struct inode **einode,
                    struct file **efile)
```

wykonuje po kolei następujące czynności:

- Tworzy nowy rekord pliku (`struct file`).
- Tworzy nowy i-węzeł (`struct inode`).
- Znajduje nieużywany indeks w tablicy deskryptorów procesu (`efd`).
- Dla i-węzła tworzy wejście katalogowe w systemie plików `eventpoll_mnt`, pod unikatową nazwą (wygenerowaną na podstawie numeru i-węzła).
- Łączy plik z wejściem katalogowym.
- Wpisuje wskaźnik do pliku pod znalezione miejsce w tablicy deskryptorów.

Funkcja

```
int ep_file_init(struct file *file, unsigned int hashbits)
```

alokuje pamięć na związany z plikiem rekord kontrolny dla *epoll* `struct eventpoll` i ustawia wskaźnik do niego w polu `private_data` rekordu `struct file`. (Parametr `hashbits` używany jest przy inicjowaniu rekordu kontrolnego). W wyniku powstaje sytuacja pokazana na rys. 4.5.

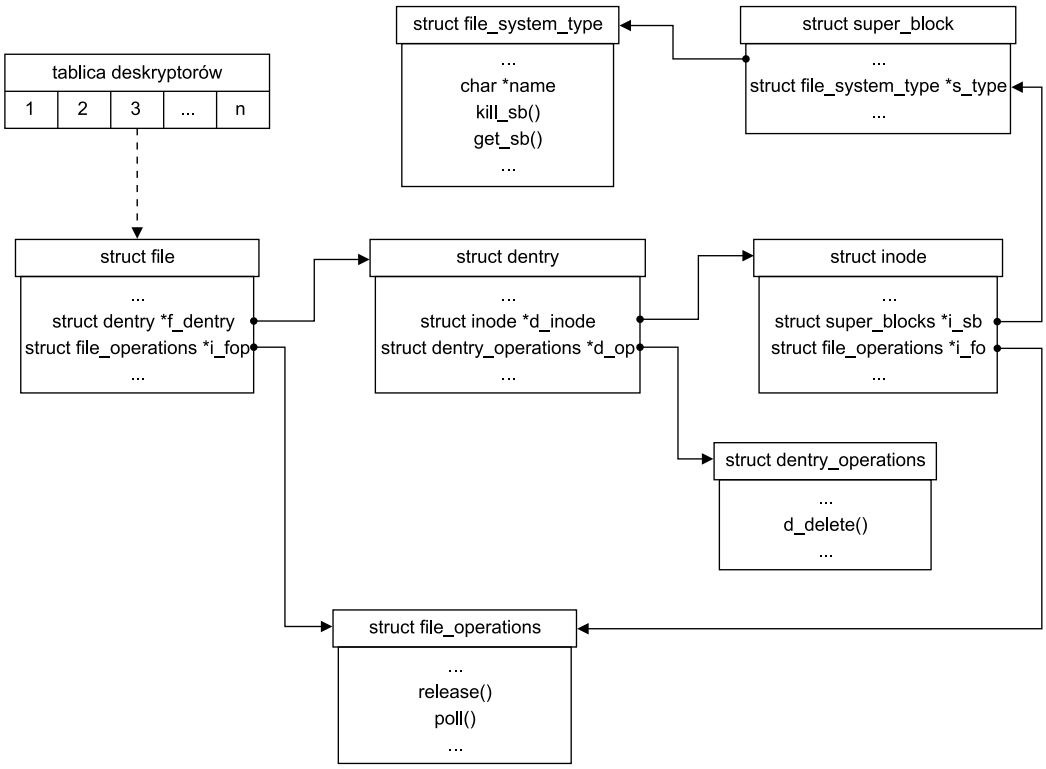
Dla systemu plików *eventpollfs* zdefiniowane są jedynie dwie operacje plikowe:

```
static struct file_operations eventpoll_fops = {
        .release        = ep_eventpoll_close,
        .poll           = ep_eventpoll_poll
}
```

- `release` — wywoływana przy zamykaniu pliku,
- `poll` — implementująca *epoll*.

Operacja `open` nie jest zaimplementowana, ponieważ nowe pliki tworzone są za pomocą funkcji `epoll_create`.

Funkcja jądra `eventpoll_exit`, wywoływana przy usuwaniu z pamięci modułu `eventpoll`, wyrejestrowuje system plików *eventpollfs* poprzez wywołanie funkcji `unregister_filesystem` oraz odmontowuje go za pomocą funkcji `mntput`.




Rysunek 4.5: System plików *eventpollfs*

### 4.4.3. Działanie *epoll*

By dodać nowy deskryptor do *epoll*, użytkownik wywołuję funkcję `epoll_ctl`, powodującą wykonanie funkcji jądra

```
long sys_epoll_ctl(int epfd, int op, int fd,
                   struct epoll_event __user *event)
```

która:

- Otwiera plik `epfd`, sprawdza czy jest on deskryptorem `epoll` i jeśli tak, to pobiera wskaźnik do jego rekordu kontrolnego.
- Otwiera plik `fd` i sprawdza czy obsługuje on operację poll.
- Kopiuje podany przez użytkownika rekord zdarzenia `event` do przestrzeni jądra.
- W zależności od parametru `op` wykonuje odpowiednią operację, np. dla dodawania pliku `EPOLL_CTL_ADD` wywoływana jest funkcja `ep_insert`.

Funkcja

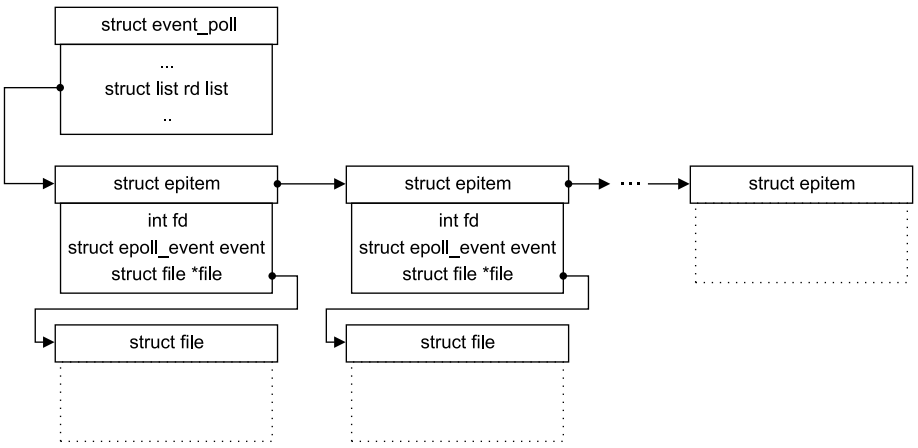
```
int ep_insert(struct eventpoll *ep, struct epoll_event *event,
              struct file *tfile, int fd)
```

rejestruje do *epoll* `ep` zdarzenia `event` zachodzące na pliku `tfile`:

- Jest tworzony i inicjowany rekord `struct epitem` powiązany z plikiem `tfile`, do przechowania zdarzeń gotowości, które będą na nim zachodzić (rys. 4.6).

- Wywoływana jest funkcja `poll` dla pliku (`tfile->f_op->poll`), której podawany jest wskaźnik do pliku oraz wskaźnik do rekordu `poll_table`, zawierający wskaźnik do procedury kolejkowej `ep_ptable_queue_proc`.

- Jeżeli plik jest gotowy, to powiązany z nim rekord zdarzeń `epitem` wstawiany jest do kolejki zdarzeń gotowych.




Rysunek 4.6: Rekordy zdarzeń `struct epitem`

Metoda pliku

```
f_op->poll(struct file *tfile, poll_table *pt)
```

odpowiada za:

- Uruchomienie procedury `pt->qproc` (jeżeli została podana) która wstawi do kolejki *epoll* pliku (budzonej przy każdej zmianie stanu) wywołanie zwrotne *epoll* (do tego celu używa się makra `poll_wait` zdefiniowanego w *include/linux/poll.h*).

- Przekazanie aktualnego stanu, w jakim znajduje się plik (gotowość do zapisu, gotowość do odczytu itp.).

Funkcja `ep_insert` jako procedura do uruchomienia na kolejce oczekiwania podaje

```
ep_ptable_queue_proc(struct file *file, wait_queue_head_t *whead,
                     poll_table *pt)
```

która wstawia funkcję zwrotną `ep_poll_callback` do wskazanej kolejki oczekiwań.

Funkcję

```
ep_poll_callback(wait_queue_t *wait, unsigned mode, int sync)
```

wywołuje się w momencie, gdy kolejka oczekiwań pliku zostanie obudzona. W wyniku jej działania:

- Jeżeli powiązane z plikiem zdarzenia `epitem` nie są jeszcze w liście gotowych, to są tam wstawiane. Powinno to następować przy każdej zmianie stanu pliku.

- Budzone są procesy oczekujące w wywołaniu funkcji `epoll_wait` (na kolejce `epoll ep->wq`) na gotowość deskryptorów. Budzenie odbywa się bez względu na to, w jakim stanie znajduje się plik. Sprawdzenie, czy stan jest odpowiedni, odbywa się poprzez wywołanie funkcji `poll` dla pliku (bez podanego wskaźnika `pt` — funkcja zwrotna dla *epoll* jest już wstawiona do odpowiedniej kolejki), przy przekazywaniu zdarzeń użytkownikowi.
- Budzona jest kolejka oczekiwań dla *epoll* `ep->poll_wait`.

W ten sposób uruchamiają się procedury zwrotne, wstawione w wyniku rejestracji deskryptora naszego *epoll* na innym *epoll*. (Jak już zostało to powiedziane — dla systemu plików *epoll* zaimplementowana jest operacja `f_op->poll`, można więc zarejestrować go na innym deskryptorze `epoll`).

### 4.4.4. Automatyczne usuwanie deskryptorów z *epoll*

Rozważmy następującą sytuację: proces rejestruje w deskryptorze *epoll* plik, po czym zamyka go, bez uprzedniego wyrejestrowania. W takim przypadku system operacyjny wyrejestruje zamykany plik z *epoll* automatycznie. By to umożliwić, każdy otwarty plik, w swoim rekordzie `file struct`, trzyma wskaźnik do listy wszystkich związanych z nim rekordów zdarzeń `struct epitem`. W momencie zamykania pliku, na początku funkcji `__fput` (zdefiniowanej w pliku *fs/file_table.c*) wywoływana jest funkcja `eventpoll_release`, która za pomocą funkcji `__eventpoll_release` usuwa zamykany plik ze wszystkich deskryptorów *epoll*, w których był on zarejestrowany.

Dzięki temu, przy implementacji *epoll* dla nowego systemu plików nie musimy implementować ręcznego usuwania naszego pliku z *epoll* przy zamykaniu.

### 4.4.5. Implementacja funkcji epoll dla nowego systemu plików

Jak wynika z wcześniejszego opisu, implementacja *epoll* dla nowego systemu plików jest bardzo prosta. Wystarczy jedynie:

- Dodać do i-węzła kolejkę oczekiwań na potrzeby `epoll` lub, jeśli jest to możliwe, użyć do tego celu istniejącej kolejki oczekiwań — tak jak ma to miejsce dla łączy (ang. *pipe*) i kolejek (ang. *fifo*), gdzie używa się kolejki (`inode.i_pipe->wait`), na której procesy czekają aż plik będzie gotowy do odczytu-zapisu.
- Zaimplementować dla pliku operację `poll (file->f_op->poll)` tak by:
  - przekazywała stan w jakim aktualnie znajduje się plik,
  - za pomocą makra `poll_wait` uruchamiała na kolejce oczekiwań podaną przez *epoll* procedurę `pt->q_proc`.
- Zaimplementować budzenie kolejki oczekiwań przy każdej zmianie stanu pliku.

W następnym rozdziale opisane jest dokładnie jak do mechanizmu *epoll* został dodany system plików, utworzony w ramach niniejszej pracy, dla kontekstów AIO.

# Rozdział 5

# Implementacja łat

## 5.1. Wstęp

W rozdziale tym opisane są stworzone w ramach niniejszej pracy dwie łaty na jądro Linuksa 2.6.1 (dla architektury i386), umożliwiające używanie AIO w połączeniu z mechanizmem powiadamiania *epoll*.

## 5.2. Wersja *zachowawcza*

Przy projektowaniu i implementacji łaty w wersji *zachowawczej* główny nacisk położony został na to, by wprowadzane modyfikacje w jądrze Linuksa były możliwie najmniejsze i zachowywały pełną zgodność z wersją oficjalną.

Mechanizm *epoll* przekazuje informacje o gotowości deskryptorów; aby włączyć do niego AIO należy w jakiś sposób skojarzyć operacje asynchroniczne z deskryptorem pliku. W tym celu w łacie *zachowawczej* do jądra Linuksa dodana została nowa funkcja systemowa:

```
long sys_io_bind(aio_context_t ctx_id).
```

Dla kontekstu operacji asynchronicznych, określonego przez identyfikator `ctx_id`, tworzy się plik, z którym będzie on związany, i przekazuje numer jego deskryptora. W przypadku gdy kontekst był już wcześniej związany z jakimś plikiem przekazywany jest błąd `EEXISTS`. Jeżeli operacja związania kontekstu z deskryptorem nie powiedzie się, to przekazywany jest odpowiedni kod błędu.

Otrzymany w ten sposób deskryptor użytkownik może za pomocą `epoll_ctl` zarejestrować na deskryptorze *epoll*. Dla deskryptora kontekstu AIO zaimplementowane zostały dwa typy zdarzeń:

- `POLLOUT` — gotowość do odczytu — na kontekście oczekują zdarzenia zakończenia wykonania operacji asynchronicznej.
- `POLLIN` — gotowość do zapisu — w ramach kontekstu można zlecić wykonanie kolejnej operacji (kontekst nie jest przepełniony).

Za pomocą funkcji `epoll_wait` można oczekiwać na gotowość kontekstu w połączeniu z oczekiwaniem na deskryptory innych plików dowolnych (implementujących *epoll*) typów.

W dalszej części rozdziału przeanalizowana jest dokładnie implementacja łaty.

### 5.2.1. Wywołanie systemowe `sys_io_bind`

Do jądra Linuksa dodane zostało nowe wywołanie systemowe `sys_io_bind`:

1. W pliku *include/asm-i386/unistd.h* została zdefiniowana stała `__NR_io_bind` przyporządkowująca naszemu wywołaniu nie używany dotąd numer 251.

2. W pliku *arch/um/kernel/sys_call_table.c* pojawiła się deklaracja funkcji `sys_io_bind` oraz w tablicy funkcji systemowych `sys_call_table` pod indeksem `__NR_io_bind` dopisany został jej adres.

3. W pliku *linux-2.6.1-jsz-standard/arch/i386/kernel/entry.S*, w tablicy funkcji systemowych `sys_call_table` dla architektury intel386 na pozycji 251 została wpisana funkcja `sys_io_bind`.

4. Definicja funkcji `sys_io_bind` znalazła się w pliku *fs/aio.c*, w którym znajduje się kod obsługujący AIO.

Funkcja `sys_io_bind` sprawdza czy podany jako argument kontekst związany jest już z jakimś deskryptorem. By to umożliwić, do rekordu `struct kioctx` opisującego kontekst AIO z poziomu jądra, dodany został wskaźnik `aio_file` na rekord `file struct`. Jeżeli kontekst jest już powiązany z jakimś plikiem przekazywany jest błąd `EEXISTS`. W przeciwnym wypadku za pomocą funkcji `aio_create_fd` tworzony jest nowy plik systemu plików *aiofs*.

### 5.2.2. Nowy system plików

By umożliwić odwoływanie się do kontekstu poprzez deskryptor, w ramach łaty dodano nowy typ systemu plików: *aiofs*. Każdy z otwartych w nim plików związany jest z kontekstem AIO. Pliki *aiofs* istnieją jedynie w pamięci operacyjnej i po zamknięciu są z niej usuwane.

System plików *aiofs* został stworzony na bazie innych systemów plików Linuksa o takich samych cechach: opisywanego w rozdziale 4.4 systemu plików *epollfs* oraz systemu plików dla łączy *pipefs*.

Montowanie, odmontowywanie oraz tworzenie nowego pliku odbywa się analogicznie jak opisano to w p. 4.4.2. Zwrócimy tu jedynie uwagę na kilka istotniejszych różnic:

- System plików rejestrowany jest (oczywiście) z inną nazwą (*aiofs*) oraz z innym magicznym numerem.

- Pole `private_data` rekordu pliku `file_struct` zawiera wskaźnik do rekordu opisującego odpowiadający mu kontekst `struct kioctx`.

- Operacja zamykania pliku zmniejsza, powiększony w trakcie wywołania `sys_io_bind`, licznik użycia kontekstu, tak by umożliwić jego usunięcie z pamięci (szczegóły w p. 5.2.4).

- By umożliwić integrację z *epoll*, dla plików *aiofs* została zaimplementowana operacja `poll`.

### 5.2.3. Integracja z *epoll*

Na potrzeby *epoll* do rekordu `struct kioctx` dodana została nowa kolejka oczekiwań `poll_wait`. Są w niej trzymane wywołania zwrotne dla *epoll*. Kolejka ta budzona jest przy każdej potencjalnej zmianie stanu pliku, czyli:

- w momencie zakończenia operacji asynchronicznej — pod koniec funkcji `aio_complete` (nowe zdarzenia zakończenia jest gotowe do pobrania),
- gdy zdarzenie zakończenia zostanie odczytane z kontekstu — po pomyślnym wykonaniu funkcji `aio_read_evt` (jest wolne miejsce na nowe zlecenie).

Dla systemu plików *aiofs* metoda pliku `poll` (por. p. 4.4) implementowana jest przez funkcję:

```
unsigned int aiofs_poll(struct file *file, poll_table *wait)
```

która:

- By wstawić wywołanie zwrotne, wywołuje na swojej kolejce oczekiwań makro `poll_wait`.
- Jeżeli w buforze kontekstu są jakieś zdarzenia, to ustawia bit `EPOLLOUT` przekazywanej wartości (gotowość kontekstu do odczytu — pobrania zdarzenia).
- Jeżeli liczba wykonujących się, to operacji w kontekście jest mniejsza niż liczba wolnych miejsc w buforze na zdarzenia ustawia bit `EPOLLIN` przekazywanej wartości (gotowość kontekstu do zapisu — zlecenia nowej operacji).

Jak już to zostało opisane w p. 4.4 w momencie zamykania pliku, epoll za pomocą funkcji `eventpoll_release` sam usuwa z kolejek oczekiwań wszystkie odnoszące się do niego wywołania zwrotne. W naszym przypadku, możliwa jest sytuacja, w której kontekst zostanie usunięty za pomocą `sys_io_destroy`, podczas gdy jego plik będzie dalej otwarty. Dlatego też dodana została funkcja

```
aiofs_release_context(struct kioctx *ioctx).
```

Wywoływana jest ona w momencie usuwania kontekstu. Jeżeli usuwany kontekst związany jest z plikiem *aiofs*, to w wyniku jej działania:

- Z rekordu pliku `file_struct` usuwany jest wskaźnik do kontekstu.
- Na pliku uruchamiana jest funkcja `eventpoll_release`, by usunąć odniesienia do niego ze wszystkich deskryptorów *epoll*, na których był zarejestrowany.
- Za pomocą `put_aioctx` zmniejszany jest licznik użycia kontekstu AIO. Zauważmy, że odniesienie do kontekstu zostało usunięte już z pliku, więc plik ten stał się plikiem *niepoprawnym* (nie związanym z żadnym kontekstem) i jego licznik użycia nie może zostać automatycznie zmniejszony przy zamykaniu.

### 5.2.4. Synchronizacja dostępu

Na początku działania funkcji `sys_io_bind` za pomocą wywołania `lookup_ioctx` zwiększany jest licznik użycia kontekstu AIO, na którym wykonywana jest operacja. Sprawdzanie, czy kontekst związany jest już z jakimś deskryptorem i ew. tworzenia dla niego nowego pliku wykonywane jest wewnątrz sekcji krytycznej kontekstu — po przejściu przez blokadę wirującą `ctx_lock`. Dzięki temu zapobiega się wyścigowi pomiędzy dwoma procesami tego samego wątku pragnącymi wykonać `sys_io_bind`.

Licznik użycia zmniejszany jest za pomocą `put_ioctx` dopiero przy ostatnim zamknięciu pliku, wewnątrz funkcji `aiofs_release`. Dopóki kontekst związany jest z plikiem, dopóty nie zostanie usunięty z pamięci.

Funkcja `aiofs_poll` sprawdza, czy kontekst zawiera informacje o zakończonych operacjach oraz czy możliwe jest zlecenie nowego zadania za pomocą operacji atomowego odczytu (makr `kmap_atomic` i `kunmap_atomic`). Użyto tutaj analogicznego rozwiązania jak wewnątrz funkcji `aio_read_evt`.

### 5.2.5. Dziedziczenie

Zwróćmy uwagę na pewien problem, który występuje przy utożsamianiu kontekstu AIO z deskryptorem. Kontekst związany jest z pamięcią operacyjną procesu, przy wywołaniu `fork` nie jest on w żaden sposób dziedziczony przez potomka, podczas gdy domyślnie dziedziczone są wszystkie deskryptory. Może to prowadzić do sytuacji, w której proces potomny będzie miał dostęp do utworzonego przez `sys_io_bind` deskryptora kontekstu, podczas gdy sam kontekst jest w wyłącznym posiadaniu procesu rodzicielskiego.

Możliwym rozwiązaniem tego problemu byłoby niedziedziczenie deskryptorów kontekstów AIO. Dokładnie tak działają deskryptory *kolejek zdarzeń* w systemie operacyjnym FreeBSD. W łacie *zachowawczej* autor nie zdecydował się na implementację tego rozwiązania, gdyż wiązałaby się ona z koniecznością wprowadzenia dość szerokich zmian w jądrze Linuksa (implementacji dziedziczenia deskryptora w zależności od typu pliku), co kłóci się z założeniem, by wprowadzać jak najmniej modyfikacje. Niedziedziczenie zostało zaimplementowane w wersji *odważnej*, gdzie było to konieczne, co opisano w p. 5.3.2.

Zauważmy, że bardzo podobny problem występuje w *epoll*. Deskryptor *epoll* jest dziedziczony pomiędzy procesami i gdy jeden z nich zarejestruje w nim nowy deskryptor, drugi będzie otrzymywał mylne zdarzenia gotowości deskryptora, którego nawet nie posiada. Ostatecznie, łata *zachowawcza* nie rozwiązuje tego problemu i podobnie jak *epoll* zakłada, że w przypadku dziedziczenia proces potomny powinien zamknąć odziedziczony deskryptor kontekstu (w przeciwnym wypadku otrzymywane przez niego informacje o gotowości będą mieć wątpliwą wartość).

## 5.3. Wersja *odważna*

Celem łaty w wersji *odważnej* było poprawienie wydajności mechanizmu obsługi operacji asynchronicznych, przy zachowaniu takiej semantyki z poziomu użytkownika, która zapewnia pełną zgodność wstecz. Przy tworzenia wersji nie starano się minimalizować wprowadzanych modyfikacji w jądrze Linuksa.

W wersji *odważnej* łaty kontekst dla operacji asynchronicznych reprezentowany jest tylko i wyłącznie przez deskryptor pliku — tak jak ma to miejsce w przypadku deskryptorów *epoll*. W momencie wywołania funkcji systemowej `io_setup` tworzony jest plik z informacjami opisującymi kontekst i przekazywany jego deskryptor. Nie ma potrzeby dodawania funkcji `sys_io_bind` służącej do kojarzenia kontekstu z deskryptorem.

Dzięki temu usunięta zostaje pewna redundancja obecna w jądrze z łatą *zachowawczą*, gdzie przy niektórych operacjach (`io_submit`) odwołujemy się do kontekstu poprzez listę z `mm_struct`, a przy innych (*epoll*) poprzez tablicę deskryptorów.

### 5.3.1. Zbędne mechanizmy z jądra Linuksa

Z rekordu `mm_struct` została usunięta lista, w której przechowywane są konteksty; wszystkie odwołania będą następować poprzez deskryptor. Zauważmy, że zmiana taka była stosunkowo łatwa do wprowadzenia, gdyż każdy kontekst przed wykonaniem na nim operacji był wyszukiwany z listy za pomocą funkcji `ioctx_lookup`, która dodatkowo zwiększała jego licznik

użycia. Wystarczyło więc zmienić kod, tak by funkcja `ioctx_lookup` brała wskaźnik do rekordu kontrolnego kontekstu z obiektu pliku odpowiedniego deskryptora (jeżeli oczywiście jest on typu *aiofs*).

W jądrze z łatą w wersji *odważnej* do zamykania kontekstu AIO, tak jak i do zamykania każdego deskryptora, można używać operacji `close`. Nie ma potrzeby stosowania do tego celu wyspecjalizowanej funkcji, takiej jak `sys_io_destroy`. Pomimo to funkcja `sys_io_destroy` dla zachowania zgodności nie została usunięta i jedynie wywołuje ona funkcję `close` dla podanego deskryptora.

W Linuksie w momencie kończenia działania procesu i zwalniania jego pamięci wywoływana jest funkcja `aio_exit`, która czeka na zakończenie działania wszystkich kontekstów z listy `ioctx_list` i potem je usuwa (dla każdego z nich jest *tak jakby* wywoływana funkcja `sys_io_destroy`). W ten sposób mamy gwarancje, że po zakończeniu działania procesu w systemie operacyjnym nie zostaną związane z nim operacje asynchroniczne. W łacie *odważnej* funkcja `aio_exit` została usunięta jako zbędna. Konteksty związane są z deskryptorami, a w momencie gdy proces kończy działanie zamykane są wszystkie jego deskryptory. Konteksty zostaną więc automatycznie usunięte w momencie zamykania deskryptorów.

### 5.3.2. Dziedziczenie

Zwróćmy uwagę na poruszany w p. 5.2.5 problem, który łączy się z reprezentacją kontekstu AIO jako deskryptora plikowego. Operacje asynchroniczne nierozerwalnie związane są z pamięcią procesu, który je zleca. Tylko i wyłącznie do jego pamięci zostaną odczytane lub z jego pamięci zostaną pobrane dane dla operacji AIO.

W łacie *odważnej* problem ten jest znacznie poważniejszy niż w *zachowawczej*. Deskryptor kontekstu pozwala nie tylko na uzyskiwanie informacji o gotowości kontekstu, ale również umożliwia zlecenie nowych operacji i pobieranie rezultatów zakończenia zleceń. Sytuacja, w której jeden proces mógłby zlecić operację asynchroniczną, odnoszącą się do jego własnej pamięci, a wykonującą się w kontekście trzymanym w pamięci innego procesu byłaby ewidentnie błędna, a wynik jej działania trudny do określenia. Z tego też powodu autor zdecydował się na wyłączenie z dziedziczenia deskryptorów kontekstów AIO, tak jak ma to miejsce w przypadku *kolejek zdarzeń* w systemie FreeBSD.

By zaimplementować *niedziedziczenie* do jądra dodana została nowa funkcja

```
is_file_aioctx(struct file * file)
```

która na podstawie wartości wskaźnika do operacji plikowych sprawdza, czy plik jest deskryptorem kontekstu. Zmodyfikowano również funkcję `copy_files`, wywoływaną wewnątrz `copy_process` (plik *kernel/fork.c*) tak by konteksty AIO nie były dziedziczone.

# Rozdział 6

# Testy

By ocenić wydajność prezentowanych w niniejszej pracy łat, przeprowadzono testy porównujące wykonanie operacji asynchronicznych w oficjalnej dystrybucji jądra systemu Linux oraz w wersjach zmodyfikowanych.

Przeprowadzono trzy serie testów:

- `aio-open_close` — w celu określenia narzutu na liniowy czas wyszukiwania kontekstu.
- `aio-stress` — w celu sprawdzenia, jak w wyniku zastosowania łat zmieniła się przepustowość AIO.
- `read_test` — by określić, jak duże mogą być zyski wynikające z wykorzystania AIO (porównanie pod bardziej ogólnym kątem wydajności operacji synchronicznych i asynchronicznych w Linuksie).

## 6.1. Środowisko testowe

Testy zostały przeprowadzone na komputerze PC z procesorem Atholon 850 MHz oraz 384 MB pamięci RAM, z zainstalowanym systemem operacyjnym Linux 2.6.1 w dystrybucji Slackware 9.1.

W trakcie działania testów nie wykonywały się żadne inne programy.

W testach `aio-stress` i `read_test` na potrzeby odczytu i zapisu specjalnie utworzono pustą partycją EXT2 o rozmiarze 2GB. Maksymalna przepustowość odczytu z testowanego dysku, mierzona komendą `hdparm`, wynosiła 26 MB/s.

## 6.2. Test `aio-open_close`

Celem testu było sprawdzenie, jak duże mogą być narzuty w jądrze Linuksa na wyszukiwanie kontekstu AIO z listy kontekstów. By to przetestować napisano prostą aplikację `aio-open_close`, której działanie polega na utworzeniu, a później zamknięciu podanej liczby kontekstów. Zauważmy, że w standardowym jądrze Linuksa nowy kontekst przy tworzeniu jest dodawany na początek listy. Dlatego też test `aio-open_close` działa w trzech trybach:

- Konteksty są tworzone i zamykane w tej samej kolejności (w standardowym jądrze, przy operacji zamykania, zamykany kontekst jest zawsze *na końcu* listy, co powoduje konieczność przeszukania jej całej).

- Konteksty są zamykane w odwrotnej kolejności, niż były tworzone (w standardowym jądrze, przy operacji zamykania, zamykany kontekst jest zawsze *na początku* listy).

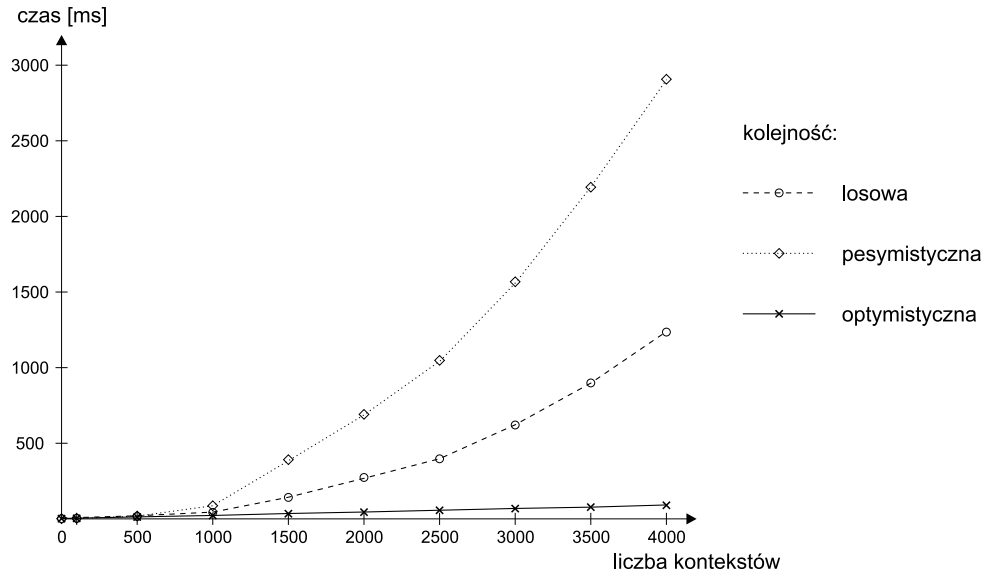
- Konteksty są zamykane w kolejności losowej.

### 6.2.1. Wyniki

Do zmierzenia czasu, jaki zajęło wykonanie programu użyto komendy `time` powłoki bash. Każdy test został powtórzony 10 razy, w poniższych tabelach podano uśrednione wyniki w milisekundach oraz odchylenie standardowe. Na wykresach pokazany jest czas w jakim program wykonywał się w przestrzeni jądra (sys) w zależności od liczby utworzonych kontekstów.

| liczba kontekstów | średni czas wykonania i odchylenie standardowe [ms] | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | kolejność optymistyczna | | | | kolejność pesymistyczna | | | | kolejność losowa | | | |
| | sys | odch. | real | odch. | sys | odch. | real | odch. | sys | odch. | real | odch. |
| 0 | **1.6** | 0.52 | 2.0 | 0.00 | **1.3** | 0.48 | 2.0 | 0.00 | **1.0** | 0.67 | 2.0 | 0.00 |
| 100 | **3.3** | 0.48 | 4.0 | 0.00 | **3.3** | 0.95 | 4.0 | 0.00 | **3.5** | 0.53 | 4.0 | 0.00 |
| 500 | **10.7** | 1.06 | 12.8 | 0.42 | **30.5** | 10.46 | 31.5 | 10.01 | **15.9** | 0.88 | 17.0 | 0.82 |
| 1000 | **22.0** | 1.25 | 23.6 | 0.52 | **155.4** | 5.06 | 157.2 | 5.16 | **67.7** | 2.41 | 69.8 | 1.81 |
| 1500 | **32.9** | 1.10 | 34.5 | 0.53 | **391.5** | 14.95 | 393.9 | 15.60 | **142.0** | 6.20 | 144.7 | 5.79 |
| 2000 | **43.6** | 1.96 | 46.3 | 0.95 | **691.1** | 19.29 | 694.0 | 19.65 | **273.7** | 7.63 | 277.3 | 7.18 |
| 2500 | **54.2** | 2.10 | 57.4 | 1.35 | **1047.9** | 20.00 | 1051.6 | 19.79 | **397.1** | 17.92 | 399.9 | 17.69 |
| 3000 | **66.1** | 2.23 | 69.6 | 1.58 | **1567.9** | 16.56 | 1572.1 | 17.02 | **620.8** | 17.08 | 624.9 | 17.85 |
| 3500 | **78.2** | 1.23 | 81.9 | 1.60 | **2193.3** | 14.64 | 2198.2 | 14.49 | **898.8** | 24.50 | 903.9 | 24.48 |
| 4000 | **88.3** | 2.31 | 92.5 | 1.65 | **2907.0** | 10.24 | 2914.2 | 8.84 | **1235.2** | 22.11 | 1240.2 | 21.72 |

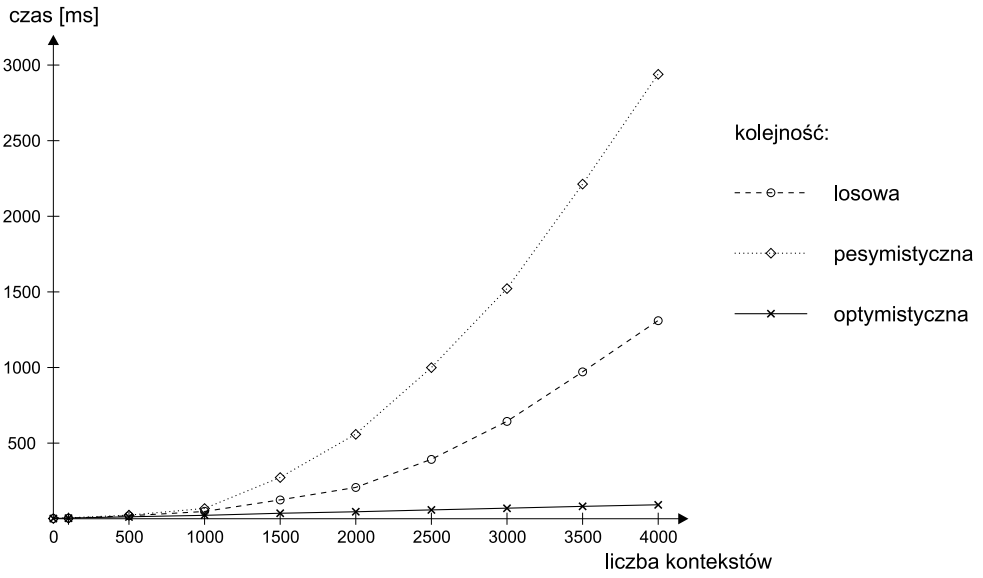
Tabela 6.1: Czas wykonania programu `aio-open_close` dla standardowego jądra Linuksa




Rysunek 6.1: Zależność czasu wykonania (sys) programu `aio-open_close` od liczby tworzonych kontekstów dla standardowego jądra Linuksa

| liczba kontek- stów | średni czas wykonania i odchylenie standardowe [ms] kolejność optymistyczna sys | odch. | real | odch. | kolejność pesymistyczna sys | odch. | real | odch. | kolejność losowa sys | odch. | real | odch. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | **1.2** | 0.67 | 2.0 | 0.00 | **1.1** | 0.88 | 2.0 | 0.00 | **1.5** | 0.71 | 2.0 | 0.00 |
| 100 | **2.9** | 0.74 | 4.0 | 0.00 | **3.6** | 0.97 | 4.0 | 0.00 | **3.3** | 0.67 | 4.0 | 0.00 |
| 500 | **11.8** | 0.79 | 13.0 | 0.00 | **31.3** | 10.03 | 32.4 | 10.12 | **16.4** | 1.07 | 17.2 | 0.63 |
| 1000 | **22.3** | 1.42 | 24.4 | 0.52 | **158.6** | 6.10 | 160.5 | 5.78 | **71.0** | 1.94 | 72.2 | 1.87 |
| 1500 | **33.0** | 1.83 | 36.0 | 0.00 | **384.8** | 33.09 | 387.2 | 32.87 | **130.4** | 7.83 | 132.6 | 7.63 |
| 2000 | **45.0** | 1.15 | 47.7 | 0.48 | **677.7** | 19.84 | 680.9 | 19.13 | **239.9** | 11.57 | 243.2 | 10.74 |
| 2500 | **55.7** | 1.42 | 59.0 | 0.47 | **1043.6** | 12.40 | 1047.9 | 10.76 | **406.8** | 13.51 | 410.4 | 12.43 |
| 3000 | **67.0** | 1.41 | 71.5 | 0.53 | **1564.8** | 11.65 | 1570.2 | 11.76 | **621.5** | 19.69 | 626.0 | 18.15 |
| 3500 | **79.5** | 1.84 | 83.4 | 0.84 | **2192.0** | 11.32 | 2196.8 | 12.58 | **898.8** | 25.01 | 903.8 | 24.83 |
| 4000 | **91.8** | 1.99 | 95.5 | 0.85 | **2912.9** | 9.90 | 2918.5 | 9.48 | **1229.0** | 27.96 | 1233.7 | 27.51 |

Tabela 6.2: Czas wykonania programu `aio-open_close` dla jądra Linuksa z łatą *zachowawczą*




Rysunek 6.2: Zależność czasu wykonania (sys) programu `aio-open_close` od liczby tworzonych kontekstów dla jądra Linuksa z łatą *zachowawczą*

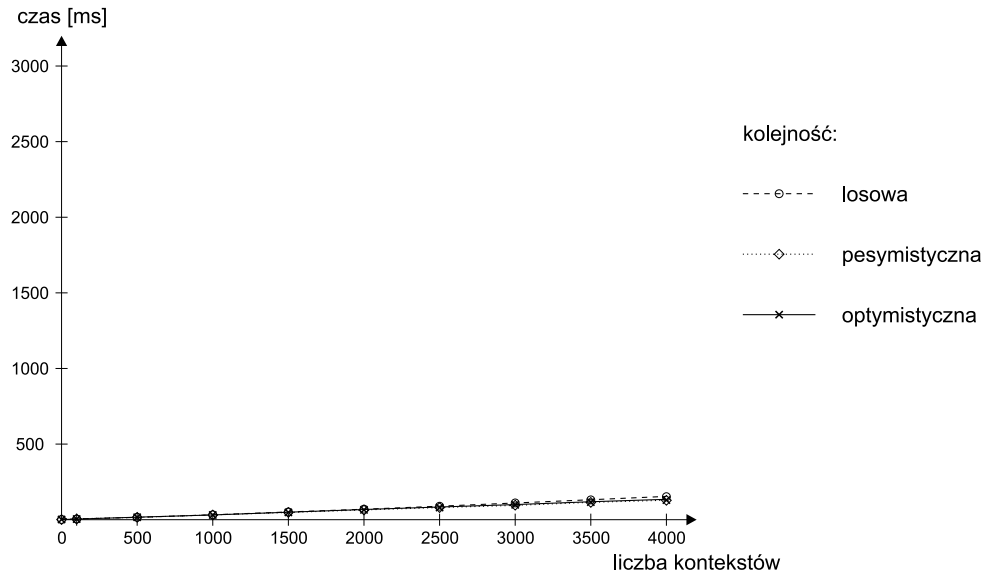
### 6.2.2. Wnioski

Test wykazał, że w standardowej dystrybucji Linuksa i w łacie w wersji *zachowawczej* czas wyszukiwania kontekstu z listy zależy liniowo od liczby otwartych przez proces kontekstów (por. rys. 6.1 i 6.2 oraz tab. 6.1 i 6.2). Zarówno przy zamykaniu kontekstów w kolejności losowej, jak i pesymistycznej czas działania programu testowego był rzędu $O(n^2)$. Czas wyszukania w oryginalnej dystrybucji i w łacie *zachowawczej* był w przybliżeniu równy.

W przypadku rozpatrywanego programu zastosowanie łaty w wersji *odważnej* przynosi poprawę o rząd wielkości (por. rys. 6.3 i 6.4 oraz tab. 6.3). Czas wyszukania jest stały, a program testowy wykonuje się w czasie rzędu $O(n)$ — niezależnie od kolejności w jakiej deskryptory są zamykane.

| liczba kontekstów | średni czas wykonania i odchylenie standardowe [ms] | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | kolejność optymistyczna | | | | kolejność pesymistyczna | | | | kolejność losowa | | | |
| | sys | odch. | real | odch. | sys | odch. | real | odch. | sys | odch. | real | odch. |
| 0 | **1.0** | 0.71 | 2.0 | 0.00 | **1.0** | 0.82 | 2.0 | 0.00 | **1.1** | 0.32 | 2.0 | 0.00 |
| 100 | **3.9** | 0.74 | 5.0 | 0.00 | **3.3** | 0.95 | 5.0 | 0.00 | **4.0** | 0.67 | 5.0 | 0.00 |
| 500 | **14.5** | 0.85 | 16.4 | 0.52 | **15.2** | 0.92 | 16.5 | 0.53 | **15.3** | 0.95 | 16.8 | 0.42 |
| 1000 | **30.0** | 1.56 | 32.4 | 0.84 | **30.2** | 0.92 | 32.3 | 0.67 | **32.2** | 1.23 | 33.7 | 0.67 |
| 1500 | **45.8** | 1.14 | 48.4 | 0.97 | **46.8** | 1.40 | 48.7 | 0.48 | **48.5** | 1.96 | 51.8 | 1.55 |
| 2000 | **61.6** | 2.07 | 65.3 | 1.42 | **61.5** | 2.27 | 64.9 | 0.99 | **69.0** | 2.40 | 71.4 | 1.65 |
| 2500 | **77.5** | 1.27 | 81.9 | 1.73 | **78.0** | 2.36 | 81.5 | 1.58 | **87.4** | 2.22 | 91.3 | 1.49 |
| 3000 | **94.3** | 1.89 | 99.3 | 1.95 | **95.8** | 1.75 | 99.2 | 1.55 | **108.3** | 2.41 | 112.4 | 2.76 |
| 3500 | **110.6** | 1.90 | 116.1 | 2.13 | **109.1** | 4.10 | 115 | 1.25 | **126.5** | 2.60 | 132.8 | 1.62 |
| 4000 | **126.5** | 2.55 | 131.7 | 1.89 | **123.8** | 1.93 | 130.9 | 1.20 | **148.9** | 4.28 | 155.1 | 4.12 |

Tabela 6.3: Czas wykonania programu `aio-open_close` dla jądra Linuksa z łatą *odważną*




Rysunek 6.3: Zależność czasu wykonania (sys) programu `aio-open_close` od liczby tworzonych kontekstów dla jądra Linuksa z łatą *odważną*

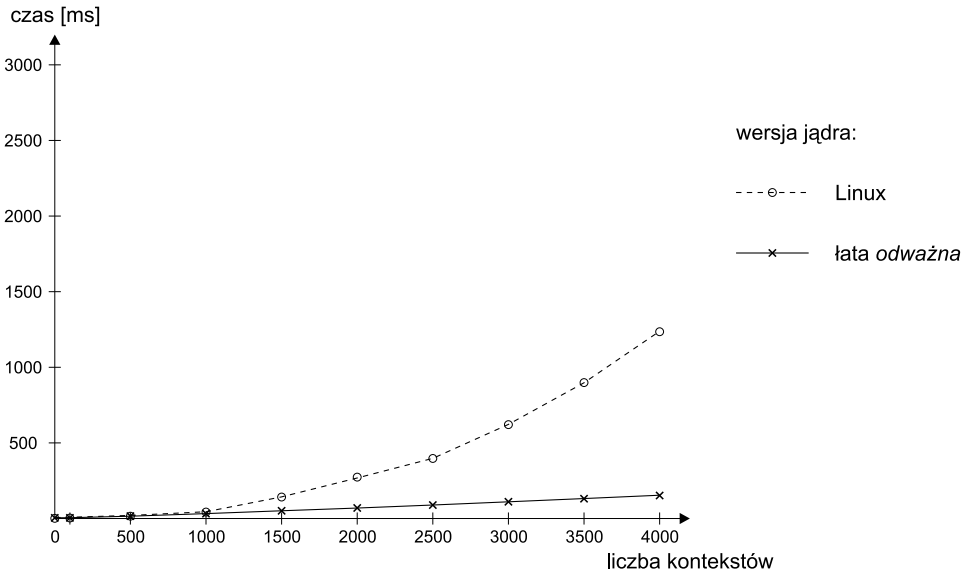
## 6.3. Test `aio-stress`

Celem testu było sprawdzenie jak wprowadzone modyfikacje wpłynęły na przepustowość operacji asynchronicznych. W tym celu użyto zmodyfikowanej wersji programu `aio-stress` autorstwa Chrisa Masona, służącego do badania wydajności operacji asynchronicznych, dostępnego w pakiecie `ext3-tools`.

Modyfikacja programu `aio-stress` polegała na wprowadzeniu dodatkowego parametru — liczby sztucznych kontekstów AIO, które zostaną utworzone. Będą się znajdować na liście przed kontekstem wykorzystywanym przez program, tak by móc sprawdzić, jak duże narzuty powoduje wyszukiwanie kontekstu.

### 6.3.1. Wyniki

Test został przeprowadzony dla pliku wielkości 64 MB, przy rozmiarze bufora dla operacji asynchronicznej równym 4 KB i maksymalnej liczbie 64 operacji asynchronicznych wykonują-




Rysunek 6.4: Porównanie czasów wykonania (sys) dla standardowego jądra Linuksa oraz wersji z łatą *odważną*

cych się jednocześnie. Liczba tworzonych sztucznych kontekstów była zmienna i wahała się od 0 do 400. Poza danymi o przepustowości, wypisywanymi przez program, za pomocą komendy `time` powłoki bash mierzony był łączny czas wykonania testu.

Testy zostały wykonane 10 razy, w tabelach prezentowane są wyniki uśrednione. Przy uśrednianiu przepustowości stosunek odchylenia standardowego do średniej nigdzie nie przekroczyło 1%. Odchylenie standardowe dla uśrednionego czasu zostało podane.

| sztuczne | przepust. zapisu [MB/s] | | przepust. odczytu [MB/s] | | średni czas wykonania [s] | | | |
|---|---|---|---|---|---|---|---|---|
| konteksty | sekwencyjnego | losowego | sekwencyjnego | losowego | sys | odch. | user | odch. |
| 0 | 16.78 | 0.54 | 17.43 | 0.53 | **1.29** | 0.023 | 495.4 | 0.51 |
| 100 | 16.76 | 0.53 | 17.44 | 0.53 | **1.38** | 0.031 | 496.4 | 1.10 |
| 200 | 16.73 | 0.54 | 17.42 | 0.53 | **1.46** | 0.017 | 495.8 | 0.73 |
| 300 | 16.76 | 0.54 | 17.44 | 0.53 | **1.54** | 0.042 | 496.4 | 0.63 |
| 400 | 16.77 | 0.54 | 17.43 | 0.53 | **1.59** | 0.040 | 495.5 | 0.87 |

Tabela 6.4: Przepustowość operacji asynchronicznych w standardowym jądrze Linuksa

| sztuczne | przepust. zapisu [MB/s] | | przepust. odczytu [MB/s] | | średni czas wykonania [s] | | | |
|---|---|---|---|---|---|---|---|---|
| konteksty | sekwencyjnego | losowego | sekwencyjnego | losowego | sys | odch. | user | odch. |
| 0 | 16.77 | 0.54 | 17.43 | 0.53 | **1.27** | 0.049 | 496.1 | 1.18 |
| 100 | 16.77 | 0.54 | 17.44 | 0.53 | **1.32** | 0.043 | 495.7 | 1.04 |
| 200 | 16.74 | 0.54 | 17.44 | 0.53 | **1.40** | 0.025 | 495.9 | 0.81 |
| 300 | 16.75 | 0.54 | 17.43 | 0.53 | **1.46** | 0.032 | 496.0 | 0.82 |
| 400 | 16.76 | 0.54 | 17.43 | 0.53 | **1.57** | 0.037 | 495.7 | 0.87 |

Tabela 6.5: Przepustowość operacji asynchronicznych w jądrze Linuksa z łatą *zachowawczą*

| sztuczne konteksty | przepust. zapisu [MB/s] sekwencyjnego | losowego | przepust. odczytu [MB/s] sekwencyjnego | losowego | średni czas wykonania [s] sys | odch. | user | odch. |
|---|---|---|---|---|---|---|---|---|
| 0 | 16.76 | 0.54 | 17.43 | 0.53 | **1.34** | 0.031 | 495.6 | 0.76 |
| 100 | 16.75 | 0.54 | 17.43 | 0.53 | **1.38** | 0.040 | 495.5 | 0.86 |
| 200 | 16.77 | 0.54 | 17.41 | 0.53 | **1.38** | 0.044 | 495.9 | 0.87 |
| 300 | 16.76 | 0.54 | 17.42 | 0.53 | **1.42** | 0.053 | 495.3 | 0.48 |
| 400 | 16.75 | 0.54 | 17.44 | 0.53 | **1.40** | 0.043 | 495.2 | 1.06 |

Tabela 6.6: Przepustowość operacji asynchronicznych w jądrze Linuksa z łatą *odważną*

### 6.3.2. Wnioski

W przypadku użytego programu badającego przepustowość narzuty spowodowane wyszukiwaniem kontekstu były zaniedbywalnie małe (por. tab. 6.4 - 6.6). Wystąpiły co prawda zauważalne różnice w wykorzystanym czasie systemowym (dochodzące do 25%), lecz nie wpłynęły one na przepustowość (300 milisekund narzutu na prawie 500 sekund łącznego czasu wykonania).

Wprowadzone w łacie *odważnej* usprawnienia nie będą więc zauważalne w przypadku aplikacji spędzających większość czasu na oczekiwanie na zakończenie operacji asynchronicznych.

## 6.4. Test `read_test`

Celem testu było określenie, jak duże mogą być zyski, wynikające z uwspółbieżnienia operacji odczytu poprzez użycie mechanizmu AIO.

By to przetestować, napisano aplikację `read_test`, która czyta w sposób synchroniczny lub asynchroniczny dane ze wskazanego pliku z pominięciem mechanizmu buforowania (plik otwierany jest z flagą `O_DIRECT`). Za pomocą parametrów wywołania ustala się również:

- łączny rozmiar danych, jakie mają być odczytane z piku,
- rozmiar pojedynczej operacji odczytu,
- czy dane mają być czytane sekwencyjnie czy losowo,
- dla odczytu asynchronicznego — maksymalną liczbę operacji asynchronicznych, które będą się wykonywać jednocześnie.

### 6.4.1. Wyniki

Test został przeprowadzony na danych rozmiaru 128 MB. Wielkość pojedynczej operacji odczytu wynosiła od 4 do 1024 KB. Przy odczycie asynchronicznym liczba równocześnie obsługiwanych zleceń wahała się od 1 do 256.

Testy został powtórzony dziesięciokrotnie, w tabelach prezentowane są wyniki uśrednione. Stosunek odchylenia standardowego do średniej nigdzie nie przekraczał 1.5%.

### 6.4.2. Wnioski

W przypadku odczytu sekwencyjnego nie została zaobserwowana praktycznie żadna różnica w przepustowości ani przy zwiększaniu stopnia współbieżności odczytu, ani przy zmianie rozmiaru pojedynczej operacji (por. tab. 6.7). Różnice przy odczycie losowym były dość

| rozmiar operacji [KB] | przepustowość odczytu synch. [MB/s] | przepust. odczytu asynch. [MB/s] liczba równoczesnych zleceń: 1 | 4 | 16 | 64 | 256 |
|---|---|---|---|---|---|---|
| 4 | 18.07 | 18.05 | 18.04 | 18.07 | 18.04 | 18.08 |
| 16 | 18.08 | 18.07 | 18.06 | 18.03 | 18.04 | 18.04 |
| 32 | 18.09 | 18.06 | 18.04 | 18.05 | 18.04 | 18.08 |
| 64 | 18.03 | 18.04 | 18.05 | 17.99 | 18.00 | 17.93 |
| 256 | 18.02 | 17.98 | 17.96 | 18.01 | 18.03 | 18.06 |
| 1024 | 17.96 | 17.99 | 18.00 | 18.01 | 18.01 | 18.01 |

Tabela 6.7: Przepustowość operacji odczytu bezpośredniego — dostęp sekwencyjny

| rozmiar operacji [KB] | przepustowość odczytu synch. [MB/s] | przepust. odczytu asynch. [MB/s] liczba równoczesnych zleceń: 1 | 4 | 16 | 64 | 256 |
|---|---|---|---|---|---|---|
| 4 | 0.46 | 0.46 | 0.48 | 0.51 | 0.55 | 0.58 |
| 16 | 1.70 | 1.70 | 1.77 | 1.88 | 2.04 | 2.18 |
| 32 | 3.09 | 3.09 | 3.21 | 3.40 | 3.67 | 3.99 |
| 64 | 5.27 | 5.27 | 5.46 | 5.71 | 6.17 | 6.70 |
| 256 | 9.14 | 9.13 | 9.88 | 10.35 | 11.45 | 11.58 |
| 1024 | 13.17 | 13.12 | 13.80 | 13.96 | 13.98 | 14.05 |

Tabela 6.8: Przepustowość operacji odczytu bezpośredniego — dostęp losowy

znaczne. Dla ustalonego rozmiaru operacji, zastosowanie AIO dawało wzrost przepustowości dochodzący do 30%, przy odchyleniu standardowym poniżej 1.5% (por. tab. 6.8).

Przepustowość 18 MB/s osiągnięta przy odczycie sekwencyjnym jest najprawdopodobniej maksymalną przepustowością, jaką można osiągnąć czytając z partycji EXT2 testowanego dysku. Przyczynę wzrostu przepustowości przy odczycie losowym należy upatrywać w umożliwieniu systemowi operacyjnemu wykonania lepszego (bardziej sekwencyjnego) szeregowania żądań do dysku. Poprawę sekwenycjności powoduje zarówno zwiększenie stopnia współbieżności odczytu (jest więcej żądań, z których system operacyjny może wybierać), jak i powiększenie rozmiaru pojedynczej operacji (żądania w obrębie jednej operacji odczytu są sekwencyjne).

# Rozdział 7

# Błędy w jądrze systemu Linux

W trakcie prac nad łatami autorowi udało się znaleźć i poprawić dwa błędy w oficjalnej dystrybucji jądra Linuksa, co jest opisane w niniejszym rozdziale.

## 7.1. Błąd w funkcji `sys_io_setup`

Funkcji systemowa `sys_io_setup` (jak już zostało to opisane w p. 4.3.4) po sprawdzeniu legalności argumentów tworzy za pomocą `ioctx_alloc` nowy kontekst i kopiuje go do pamięci użytkownika poprzez wywołanie `put_user`. W sytuacji, gdy uda się utworzyć nowy kontekst (wywołanie `ioctx_alloc` zakończy się sukcesem), lecz nie uda się skopiować jego identyfikatora do pamięci użytkownika (wywołanie `put_user` zakończy się porażką) kontekst jest niszczony za pomocą `io_destroy`.

Usuwająca kontekst funkcja `io_destroy` w *normalnej* sytuacji wołana jest wewnątrz funkcji `sys_io_destroy` (odpowiednika funkcji systemowej `io_destroy`). Jeżeli kontekst, który ma być usunięty, zaznaczony jest jako *żywy*, to oznacza się go jako *martwy* oraz zmniejsza o dwa jego licznik użycia:

- Jedno zmniejszenie wynika stąd, że w początkowej fazie działania funkcji `sys_io_destroy`, przy wyszukiwaniu kontekstu za pomocą `lookup_ioctx`, jego licznik użycia jest zwiększany o jeden.

- Jedno zmniejszenie jest po to, by umożliwić usunięcie kontekstu. Kontekst tworzony jest z licznikiem ustawionym na jeden i dopóki jest on żywy, dopóty licznik nie może spaść poniżej tej wartości.

Po stworzeniu nowego kontekstu za pomocą `ioctx_alloc` wewnątrz `sys_io_setup` jego licznik użycia ustawiony jest na jeden. Wywołanie w tym momencie funkcji `io_destroy` i próba zmniejszenia go o dwa powodowała niespełnienie asercji (licznik użycia większy równy od zera) i błąd jądra.

Autor niniejszej pracy opracował łatę naprawiająca ten błąd poprzez zwiększanie licznika użycia kontekstu o jeden przed wywołaniem `io_destroy`, w momencie gdy funkcja `put_user` kończy się niepowodzeniem.

Opis błędu, wraz z przykładowym programem, w wyniku działania którego zostaje on ujawniony oraz z łatą został wysłany na listę dyskusyjną [linux-aio]. Po pozytywnym zaopiniowaniu przez Andrew Mortona i Linusa Torvaldsa wysłana łata stała się częścią jądra Linuksa począwszy od wersji 2.6.7.

## 7.2. Wyciek pamięci w *event-poll*

Przy tworzeniu nowego deskryptora *epoll*, w funkcji `epoll_create`, popełniony został poważny błąd skutkujący wyciekiem pamięci. Licznik użycia rekordu wejścia katalogowego (`struct dentry`) był niepoprawnie ustawiany na dwa:

- Wewnątrz funkcji `new_inode` wywoływanej przez `ep_eventpoll_inode` licznik użycia wejścia katalogowego ustawiany jest na jeden.
- Funkcja `ep_getfd` przy ustawianiu wskaźnika `f_dentry` w rekordzie pliku (`struct file`) niepoprawnie zwiększała wskaźnik użycia o jeden za pomocą funkcji `dget`.

Skutkiem tego powstawał rekord wejścia katalogowego (`struct dentry`) z ustawionym licznikiem użycia na dwa, podczas gdy faktycznie była trzymana do niego referencja tylko w jednym rekordzie pliku (`struct file`).

Licznik użycia rekordu wejścia katalogowego zmniejszany jest tylko wtedy, gdy zamykany jest wskazujący na niego rekord pliku. Wejście katalogowe usuwane jest z pamięci gdy jego licznik użycia spadnie do zera. W przypadku plików opisujących deskryptor *epoll* nie następowało to **nigdy**, co dla każdego utworzonego deskryptora powodowało pozostawanie w pamięci operacyjnej na zawsze odpowiadających mu rekordów wejścia katalogowego (`struct dentry`) i i-węzła (`struct inode`).

Zaistniały wyciek pamięci był dość poważny — ok. 0.5 KB na jednym deskryptorze *epoll*. Umożliwiał on praktycznie każdemu użytkownikowi zawieszenie systemu operacyjnego poprzez utworzenie i zamknięcie dużej liczby deskryptorów *epoll* (co powodowało zajęcie całej pamięci operacyjnej). Mogło być to wykorzystane przy atakach typu *odmowa usługi* (ang. *Denial of Service*).

Sporządzony przez autora opis błędu i łata, która go naprawia, zostały wysłane na listę dyskusyjną [linux-fsdevel]. Po pozytywnym zaopiniowaniu przez Davida Libenzi, Andrew Mortona i Linusa Torvaldsa łata stała się częścią jądra Linuksa począwszy od wersji 2.6.8.

# Rozdział 8

# Podsumowanie

Podstawowy cel pracy, jakim było rozszerzenie funkcjonalności jądra systemu Linux, został osiągnięty. Udało się opracować łaty integrujące AIO z mechanizmem *event-poll*. Dzięki nim proces może równocześnie oczekiwać na zakończenie zleconych operacji asynchronicznych i gotowość plików do odczytu-zapisu.

Łata w wersji *odważnej*, poprzez reprezentację kontekstu AIO jako deskryptora pliku, przyniosła znaczącą poprawę wydajności. Zredukowała ona czas wyszukania kontekstu po stronie jądra z liniowego ($O(n)$) do stałego ($O(1)$).

W przyszłości można kontynuować pracę poprzez publikację opracowanych łat na listach dyskusyjnych [linux-aio] i [linux-fsdevel] oraz zabieganie o to, by stały się one częścią oficjalnego jądra lub przynajmniej liczyć na to, że w jakimś stopniu posłużą one za źródło inspiracji dla twórców Linuksa.

Ciekawie przedstawiają się perspektywy ewolucji mechanizmu deskryptorów plików w jądrze systemu Linux. Reprezentacja kolejki oczekiwań *event-poll* jako deskryptora pliku sugeruje rozwój w kierunku rozwiązań z systemu FreeBSD. W systemie FreeBSD deskryptory są ogólnym mechanizmem, za pomocą którego proces może odwoływać się do obiektów jądra, a deskryptory plików są szczególnym przypadkiem deskryptorów. Rozwiązanie takie wymaga odejścia od dziedziczenia deskryptorów pewnych typów przez procesy potomne — krokiem w tym kierunku jest prezentowana łata *odważna*.

Za niewątpliwy sukces należy uznać znalezienie dwóch błędów w jądrze systemu Linux i opracowanie poprawiających je łat, które obecnie stanowią część standardowej dystrybucji.

Osobnym, wartym zbadania tematem jest porównanie wydajności operacji synchronicznych i asynchronicznych w Linuksie. W niniejszej pracy przeprowadzona została jedna seria takich testów. W przyszłości można przeprowadzić kolejne, bardziej rozbudowane testy, by dokładniej odpowiedzieć na pytanie, kiedy należy stosować operację asynchroniczne oraz jak duże mogą być zyski wynikające z ich wykorzystania.

# Dodatek A

# Opis załączonej płyty CD

Zawartość załączonej płyty CD:

- `linux-2.6.1`
  - `linux-2.6.1.tar.bz2` — oryginalne źródła jądra Linuksa w wersji 2.6.1.
- `patch-jsz-a`
  - `patch-2.6.1-jsz-a.diff` — łata *zachowawcza* na jądro Linuksa 2.6.1.
  - `linux-2.6.1-jsz-a.tar.bz2` — kompletne źródła jądra Linuksa 2.6.1 z łatą *zachowawczą*.
- `patch-jsz-b`
  - `patch-2.6.1-jsz-b.diff` — łata *odważna* na jądro Linuksa 2.6.1.
  - `linux-2.6.1-jsz-b.tar.bz2` — kompletne źródła jądra Linuksa 2.6.1 z łatą *odważną*.
- `bug-aio`
  - `patch-2.6.7-rc2-fixed_aiobug.diff` — łata na jądro Linuksa 2.6.7-rc2 poprawiająca błąd w funkcji systemowej `sys_io_create`.
  - `aiobug.s` — przykładowy program w assemblerze ujawniający błąd w funkcji `sys_io_create`.
  - `ChangeLog-2.6.7` — rejestr z opisem zmian pomiędzy wersjami 2.6.6 oraz 2.6.7 jądra Linuksa. Zawiera informację o dołączeniu łaty poprawiającej błąd w `sys_io_create`.
- `bug-epoll`
  - `patch-2.6.7-fixed_epollbug.diff` — łata na jądro Linuksa 2.6.7, likwidująca wyciek pamięci w *epoll*.
  - `epoll-leak.c` — przykładowy program w C ujawniający wyciek pamięci w *epoll*.
  - `ChangeLog-2.6.8` — rejestr z opisem zmian pomiędzy wersjami 2.6.7 oraz 2.6.8 jądra Linuksa. Zawiera informację o dołączeniu łaty poprawiającej wyciek pamięci w *epoll*.

# Bibliografia


[aio] S. Bhattacharya, *AIO Design Notes*, 2002, `http://lse.sourceforge.net/io/aionotes.txt`.

[bio] S. Bhattacharya, *BIO Design Notes*, 2001, `http://lse.sourceforge.net/io/bionotes.txt`.

[asl25] S. Bhattacharya, S. Pratt, B. Pulavarty, J. Morgan, *Asynchronous I/O Support in Linux 2.5*, Proceedings of the Linux Symposium, Ottawa, 23-26 czerwca 2003.

[linkern] D. P. Bovet, M. Cesati, *Linux Kernel*, Wydawnictwo RM, Warszawa 2001.

[coek] W. A. Coekaerts, *Big Servers — 2.6 compared to 2.4*, Proceedings of the Linux Symposium, Ottawa, 21-24 czerwca 2004.

[gammo] L. Gammo, T. Brecht, A. Shukla, D. Pariag, *Comparing and Evaluating epoll, select, and poll Event Mechanisms*, Proceedings of the Linux Symposium, Ottawa, 21-24 czerwca 2004.

[c10k] D. Kegel, *The C10K problem*, `http://www.kegel.com/c10k.html`.

[lemon] J. Lemon, *Kqueue: A generic and scalable event notification facility for FreeBSD*, FREENIX Track: 2001 USENIX Annual Technical Conference, Massachusetts, 25-30 czerwca 2001.

[linux-aio] Lista dyskusyjna linux-aio@kvack.org poświęcona obsłudze asynchronicznego wejścia-wyjścia w jądrze Linuksa. Archiwa dostępne za pośrednictwem `http://marc.theaimsgroup.com`.

[linux-fsdevel] Lista dyskusyjna linux-fsdevel@vger.kernel.org poświęcona systemom plików w jądrze Linuksa. Archiwa dostępne za pośrednictwem `http://marc.theaimsgroup.com`.

[msdn] *The Microsoft Developer Network*, `http://msdn.microsoft.com`.

[posix] *POSIX Technical Standard*, Issue 6, 2000.

[vah] U. Vahalia, *Jądro systemu UNIX, nowe horyzonty*, Wydawnictwo Naukowo-Techniczne, Warszawa 2001.

[linux] Źródła jądra systemu Linux w wersji 2.6.1, dostępne za pośrednictwem `http://lxr.linux.no`.