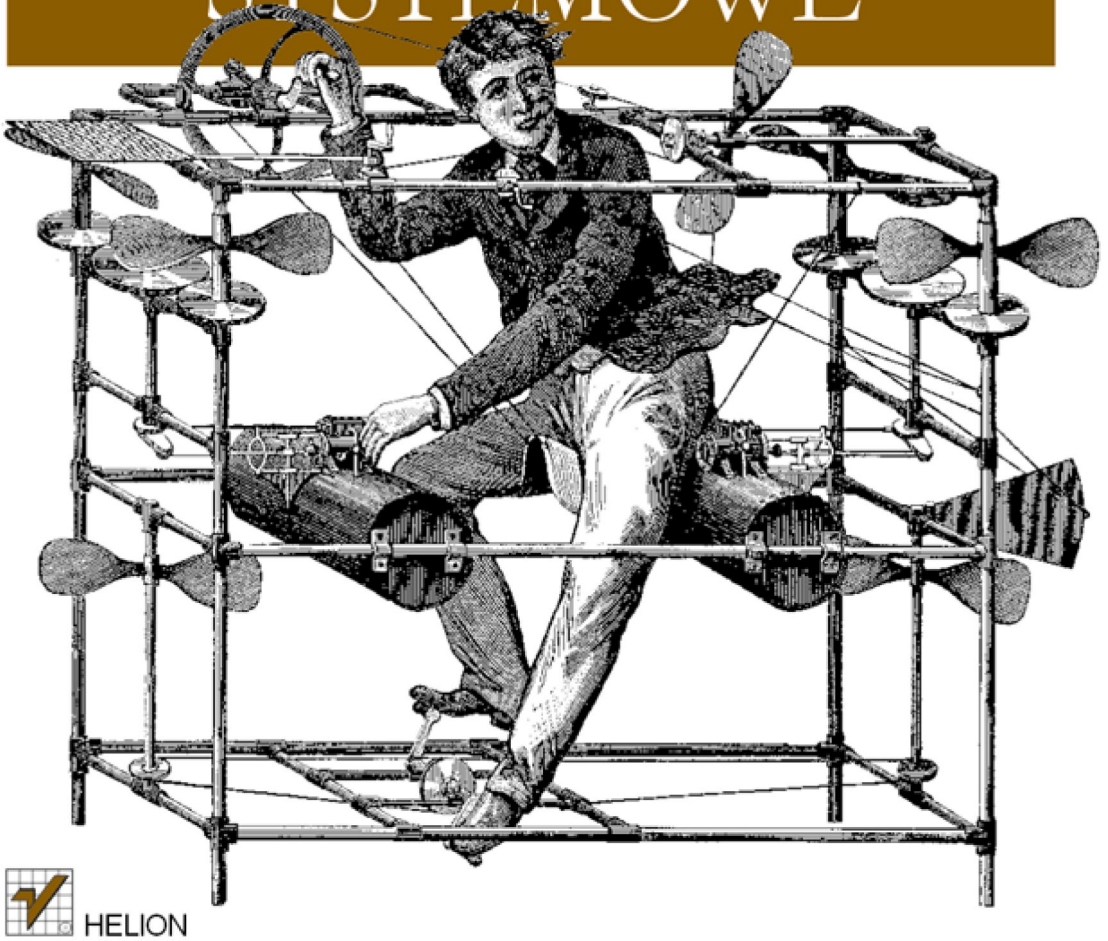
WYKORZYSTAJ MOC LINUKSA I TWÓRZ FUNKCJONALNE OPROGRAMOWANIE SYSTEMOWE!

LINUX

PROGRAMOWANIE SYSTEMOWE



HELION

O'REILLY®

ROBERT LOVE

Tytuł oryginału: Linux System Programming
ISBN: 978-83-246-6111-4

Niech idą

w PiSdu!

Spis treści

Przedmowa

Istnieje pewne zdanie, które lubią przytaczać projektanci jądra Linuksa, gdy są w złym humorze: „Przestrzeń użytkownika służy jedynie do przeprowadzania testów obciążenia jądra".

Cytując to zdanie, projektanci jądra mają na myśli odmówienie poniesienia odpowiedzialności za jakiekolwiek błędy, które przeszkodzą w poprawnym wykonaniu kodu z przestrzeni użytkownika. Jeśli chodzi o projektantów kodu z przestrzeni użytkownika, powinni oni po prostu w pokorze poprawić swoje oprogramowanie, ponieważ żaden pojawiający się problem nie jest wynikiem błędnego działania jądra, lecz rezultatem uruchomienia niewłaściwego kodu.

Aby udowodnić, że zazwyczaj to nie jądro jest winne powstałym błędom, wiodący projektant jądra Linuksa od ponad trzech lat wygłasza w przepełnionych salach konferencyjnych referat *Dlaczego przestrzeń użytkownika jest do kitu*. W swoim wystąpieniu przytacza przykłady horrendalnie zaprojektowanych kodów z przestrzeni użytkownika, na których każdy codziennie bezgranicznie polega. Inni projektanci jądra stworzyli narzędzia pokazujące, w jaki sposób programy z przestrzeni użytkownika nieprawidłowo wykorzystują sprzęt oraz zużywają baterie w laptopach.

Jednak i oni są na co dzień zależni od tego kodu. Gdyby kod z przestrzeni użytkownika nie istniał, całe oprogramowanie jądra byłoby przydatne jedynie do wyświetlania na ekranie naprzemiennych ciągów znaków ABABAB.

Linux jest obecnie najbardziej uniwersalnym oraz najpotężniejszym systemem operacyjnym, jaki kiedykolwiek powstał. Ma wielorakie zastosowanie. Poczynając od telefonów komórkowych oraz urządzeń wbudowanych, a kończąc na ponad 70% z pięciuset największych superkomputerów na świecie. Żaden inny system operacyjny nie jest tak skalowalny i nie spotyka tylu wyzwań związanych z różnymi rodzajami sprzętu i środowiska, co Linux.

Również oprogramowanie z przestrzeni użytkownika może być uruchamiane na wszystkich platformach, na których poprawnie działa kod jądra. Udostępniane są prawdziwe aplikacje i narzędzia.

Robert Love w niniejszej książce podjął się zadania nie do pozazdroszczenia. Chce przekazać czytelnikowi wiedzę na temat prawie wszystkich funkcji systemowych w Linuksie. Ta publikacja pozwoli w pełni zrozumieć, w jaki sposób z perspektywy przestrzeni użytkownika działa jądro oraz jak można spożytkować potęgę systemu.

Informacje, przedstawione w książce, pomogą w tworzeniu kodu, który będzie działać we wszystkich dystrybucjach Linuksa oraz na każdego rodzaju sprzęcie. Pozwolą również zrozumieć, jak działa ten system operacyjny i jak można wykorzystać jego uniwersalność.

Wreszcie, rzecz najważniejsza — poradnik uczy, w jaki sposób powinno tworzyć się poprawne oprogramowanie.

Greg Kroah-Hartman

Wstęp

Niniejsza książka dotyczy programowania systemowego — dokładniej rzecz ujmując, programowania systemowego w Linuksie. *Programowanie systemowe* jest procedurą tworzenia *oprogramowania systemowego*, będącego niskopoziomowym kodem komunikującym się bezpośrednio z jądrem oraz głównymi bibliotekami systemowymi. W tej książce poddane zostaną analizie wywołania systemowe Linuksa oraz inne niskopoziomowe funkcje, pochodzące na przykład z biblioteki języka C.

Wiele podręczników podejmuje tematykę programowania systemowego dla Uniksa, ale tylko kilka z nich dotyczy samego systemu Linux. Jeszcze mniej publikacji (jeśli w ogóle jakiekolwiek) zajmuje się jego najnowszymi wersjami oraz zaawansowanymi interfejsami, dedykowanymi tylko jemu. Ponadto, książka ta posiada dodatkową zaletę: osobiście stworzyłem wiele fragmentów kodu w systemie Linux, przeznaczonych zarówno dla jądra, jak również będących oprogramowaniem systemowym. W rzeczywistości zaimplementowałem niektóre funkcje systemowe oraz inne zagadnienia opisane w tej książce. Zgodnie z tym, zawarta jest w niej specjalna wiedza, informująca nie tylko o tym, w jaki sposób *powinny* działać interfejsy systemowe, lecz jak *w rzeczywistości* funkcjonują oraz w jaki, najbardziej efektywny sposób można ich używać. Książka ta łączy w sobie przewodnik po programowaniu systemowym w Linuksie, podręcznik z opisem jego funkcji systemowych oraz zaawansowany poradnik opisujący tworzenie bardziej inteligentnego i szybszego kodu. Została ona napisana w sposób przyjazny i nawet, jeśli jej czytelnikami będą programiści, którzy codziennie zajmują się tworzeniem oprogramowania na poziomie systemowym, to i tak odnajdą tu porady, pozwalające na uzyskanie lepszego kodu.

Przyjęte założenia, dotyczące odbiorców książki

Podczas tworzenia tej książki przyjęto założenie, że czytelnik posiada umiejętność programowania w języku C i zna środowisko programistyczne Linuksa — nie musi być to wiedza zaawansowana, lecz ogólna znajomość tematów. Jeśli czytelnik nie zapoznał się jeszcze z żadnymi publikacjami dotyczącymi języka programowania C, takimi jak książka Briana W. Kernighana i Dennisa M. Ritchie'go *Język C* (opublikowana przez wydawnictwo WNT), znana w skrócie jako K&R, wówczas zalecane jest przeczytanie tego typu pozycji książkowej. W przypadku, gdy czytelnik nie czuje się pewnie podczas używania uniksowych edytorów tekstowych (takich jak *Emacs* czy *vim*, które są najbardziej znane), wówczas powinien poświęcić więcej czasu na związane z nimi

ćwiczenia praktyczne. Konieczne jest również zapoznanie się z podstawami używania narzędzi *gcc*, *gdb*, *make* itd. Istnieje wiele książek dotyczących narzędzi i metod tworzenia oprogramowania w Linuksie; bibliografia, podana na końcu książki, zawiera przydatne pozycje.

Przyjęto kilka założeń dotyczących wiedzy czytelnika na temat programowania systemowego w Uniksie lub Linuksie. Książka ta rozpoczyna się od zaprezentowania podstaw, a tematy w niej poruszane, stają się coraz bardziej zaawansowane, podczas gdy omawiane zostają skomplikowane interfejsy oraz sztuczki optymalizacyjne. Każdy z czytelników, posiadających różne poziomy wiedzy, powinien odkryć w tej książce nieznane mu jeszcze informacje. Ja na pewno nauczyłem się wiele podczas jej pisania.

Nie przyjęto żadnych założeń dotyczących motywacji oraz przekonań czytelnika. Na pewno książka ta jest przeznaczona dla inżynierów, pragnących programować lepiej na niższym poziomie. Również programiści, tworzący oprogramowanie wyższego poziomu i chcący pewniej czuć się w dziedzinach, którymi się zajmują, także znajdą wiele interesujących tematów dla siebie. Lektura usatysfakcjonuje także hakerów. Niezależnie od tego, czego potrzebują lub wymagają czytelnicy, zagadnienia, przedstawione w tej książce, są na tyle różnorodne (przynajmniej te dotyczące programowania systemowego w Linuksie), że powinny spełnić ich oczekiwania.

Bez względu na wszystko, *bawcie się dobrze*!

Zawartość książki

Książka ta podzielona jest na 10 rozdziałów i zawiera dwa dodatki.

Rozdział 1. *Wprowadzenie — podstawowe pojęcia*
: Rozdział ten jest wprowadzeniem pozwalającym na zapoznanie się z systemem Linux, programowaniem systemowym, jądrem, bibliotekami języka C oraz kompilatorem C. Powinni go przeczytać nawet zaawansowani użytkownicy.

Rozdział 2. *Plikowe operacje wejścia i wyjścia*
: W rozdziale tym wprowadzono pojęcie plików, które są najważniejszą abstrakcją środowiska uniksowego. Omówiono również plikowe operacje wejścia i wyjścia, czyli podstawy programowania w Linuksie. Zaprezentowano czytanie i zapisywanie do plików, jak również inne podstawowe operacje plikowe wejścia i wyjścia. Rozdział kończy się analizą, w jaki sposób w jądrze Linuksa są zaimplementowane i obsługiwane pliki.

Rozdział 3. *Buforowane operacje wejścia i wyjścia*
: W rozdziale tym analizie zostaje poddany problem związany z interfejsami podstawowych operacji wejścia i wyjścia — zarządzanie rozmiarem bufora. Omówione zostają rozwiązania dotyczące w ogólności buforowanych operacji wejścia i wyjścia, a w szczególności standardowych operacji wejścia i wyjścia.

Rozdział 4. *Zaawansowane operacje plikowe wejścia i wyjścia*
: W rozdziale skupiono się na analizie zaawansowanych interfejsów wejścia i wyjścia, odwzorowań w pamięci oraz technik optymalizacyjnych. Omówiono przykłady niepożądanych operacji wyszukiwania, a także rolę zarządcy operacji wejścia i wyjścia w jądrze Linuksa.

Rozdział 5. *Zarządzanie procesami*
: W rozdziale tym wprowadzono pojęcie drugiej pod względem ważności abstrakcji w systemie Unix — *procesu*, a także przedstawiono rodzinę funkcji systemowych, pozwalających na podstawowe zarządzanie procesami, włącznie z funkcją *fork*.

Rozdział 6. *Zaawansowane zarządzanie procesami*
: W tym rozdziale zaprezentowano zaawansowane zarządzanie procesami; włącznie z procesami czasu rzeczywistego.

Rozdział 7. *Zarządzanie plikami i katalogami*
: W rozdziale poddano analizie operacje tworzenia, przenoszenia, kopiowania i usuwania plików oraz katalogów, a także inne, związane z nimi działania.

Rozdział 8. *Zarządzanie pamięcią*
: W rozdziale omówiono zarządzanie pamięcią. Rozpoczyna się on od przedstawienia koncepcji związanych z pamięcią, używanych w systemie Unix, takich jak przestrzeni adresowej procesu oraz stronicowania. W dalszej części rozdziału przeprowadzono analizę interfejsów, pozwalających na przydzielanie i zwalnianie pamięci jądra. Rozdział ten kończy się omówieniem zaawansowanych interfejsów, związanych z obsługą pamięci.

Rozdział 9. *Sygnały*
: W rozdziale tym zaprezentowano sygnały. Przeprowadzono analizę sygnałów oraz ukazano ich rolę w systemie Unix. Przedstawiono również interfejsy sygnałowe, poczynając od najprostszych, a kończąc na zaawansowanych.

Rozdział 10. *Czas*
: W rozdziale poddano analizie takie zagadnienia jak czas, tryb uśpienia, a także zarządzanie zegarami. Omówiono interfejsy podstawowe, zegary POSIX, a także liczniki o wysokiej dokładności.

Dodatek A *Rozszerzenia kompilatora GCC dla języka C*
: W dodatku omówiono wiele optymalizacji, udostępnianych przez *gcc* oraz GNU C, takich jak atrybuty pozwalające na zdefiniowanie funkcji stałych, czystych lub wplatanych.

Dodatek B *Bibliografia*
: Bibliografia zawiera publikacje będące użytecznym uzupełnieniem tej książki, jak również pozycje omawiające tematy, które nie zostały tutaj przedstawione.

Wersje uwzględnione w tej książce

Interfejs systemowy Linuksa jest zdefiniowany jako interfejs binarny aplikacji (ABI) oraz interfejs programowania aplikacji (API). Jest on udostępniany przez trzy elementy systemu: jądro Linuksa (będące „sercem" systemu operacyjnego), bibliotekę GNU C (*glibc*) oraz kompilator GNU C (*gcc* — obecnie nazywany formalnie GNU Compiler Collection, lecz dla potrzeb tej książki wykorzystywany jest tylko jego kompilator języka C). W tej książce zostaje poddany analizie interfejs systemowy, zdefiniowany dla jądra Linuksa w wersji 2.6.22, biblioteki *glibc* w wersji 2.5 oraz kompilatora *gcc* w wersji 4.2. Zaprezentowane tutaj interfejsy powinny być wstecznie kompatybilne z ich starszymi wersjami (z wyłączeniem nowych interfejsów), a kompatybilne w przód z wersjami nowszymi.

Jeśli wyobrazimy sobie rozwijający się system operacyjny jako ruchomy cel, wówczas Linux mógłby być reprezentowany przez geparda. Kolejne kamienie milowe, określające jego postęp, pojawiają się co parę dni, a nie lat. Często publikowane coraz to nowsze wersje jądra oraz innych komponentów wciąż zmieniają sytuację na polu gry. W żadnej książce nie można w pełni przedstawić tak dynamicznie zachowującego się systemu, jednocześnie uwzględniając wszystkie zmiany w międzyczasie w nim zachodzące.

Bez względu na to, środowisko programistyczne, zdefiniowane przez programowanie systemowe, jest *niezmienne*. Programiści jądra zadają sobie wiele trudu, aby nie dopuścić do zmian funkcji systemowych, projektanci biblioteki *glibc* cenią sobie wysoko kompatybilność wstecz i w przód, natomiast zestaw narzędzi Linuksa generuje kod, który jest kompatybilny dla różnych wersji (szczególnie w przypadku języka C). Zgodnie z tym, podczas gdy sam Linux wciąż się rozwija, programowanie systemowe w Linuksie pozostaje niezmienne, a książka, oparta na migawce systemu, wykonanej zwłaszcza w miejscu jego projektowania, posiada wciąż taką samą wartość. To, co zamierzam powiedzieć, jest proste: nie martwicie się o zmiany interfejsów systemowych i *dokonajcie zakupu tej książki*!

Użyte konwencje

W książce zostały użyte następujące konwencje typograficzne:

Czcionka pochyła
: Użyta w celu wyróżnienia tekstu oraz prezentacji nowych terminów, adresów URL, zwrotów obcojęzycznych oraz nazw narzędzi Uniksa, nazw plików, katalogów, a także ścieżek.

`Czcionka o stałej szerokości`
: Oznacza pliki nagłówkowe, polecenia, zmienne, atrybuty, funkcje, typy, parametry, obiekty, makra oraz inne konstrukcje programistyczne.

`Czcionka pochyła o stałej szerokości`
: Oznacza tekst (na przykład element ścieżki), który powinien zostać zastąpiony wartością dostarczoną przez użytkownika.

Ta ikona oznacza wskazówkę, sugestię lub uwagę ogólną.

Większość kodu, umieszczonego w tej książce, przedstawiona została w postaci zwięzłych, lecz mimo to nadających się do wykorzystania fragmentów programów. Są one podobne do poniżej zaprezentowanego kodu:

```
while (1)
{
   int ret;

   ret = fork ( );
   if (ret == -1)
      perror ("fork");
}
```

Wiele wysiłku włożono, aby udostępnić takie fragmenty kodów, które są zwięzłe, lecz mimo to nadają się do wykorzystania. Nie jest wymagane użycie specjalnych plików nagłówkowych pełnych dziwnych makr i nieczytelnych skrótów. Zamiast publikowania kilku olbrzymich programów, książka ta zawiera wiele prostych przykładów. Mam nadzieję, że będą one użyteczne podczas pierwszej lektury, a podczas kolejnych posłużą jako dobre odnośniki.

Prawie wszystkie przykłady w tej książce są samodzielne. Oznacza to, że można je w prosty sposób skopiować do edytora tekstu i następnie uruchomić. O ile nie zostanie to zaznaczone, wszystkie fragmenty kodów powinny być budowane bez użycia specjalnych znaczników kompilatora (w kilku przypadkach konieczne będzie linkowanie specjalnej biblioteki). W celu skompilowania pliku źródłowego, zaleca się wykonanie poniżej przedstawionego polecenia:

```
$ gcc -Wall -Wextra -O2 -g -o snippet snippet.c
```

Wykonanie powyższego polecenia spowoduje skompilowanie pliku źródłowego *snippet.c* do wykonywalnego pliku binarnego *snippet*, włączając wiele ostrzegawczych kontroli, znaczące lecz sensowne optymalizacje oraz wspomaganie debugowania. Kod, przedstawiony w książce, powinien zostać skompilowany za pomocą tego polecenia bez generowania błędów lub ostrzeżeń — wcześniej dla danego fragmentu kodu należy zbudować szkielet programu.

Gdy w danym podrozdziale zaprezentowana zostaje nowa funkcja, jest przedstawiona za pomocą specjalnie pogrubionej czcionki, podobnie jak ma to miejsce w przypadku uniksowego pliku pomocy:

```
#include <fcntl.h>

int posix_fadvise (int fd, off_t pos, off_t len, int advice);
```

Po zaprezentowaniu plików nagłówkowych oraz innych niezbędnych definicji, podawany jest pełny prototyp danej funkcji.

Używanie przykładowych kodów

Książka ta ma za zadanie wspomóc czytelnika, by mógł z sukcesem realizować swoje zadania. Kod z tej książki może w zasadzie być używany w jego własnych programach i dokumentacji. Nie ma potrzeby, aby kontaktować się z wydawnictwem w celu uzyskania zezwolenia na takie użytkowanie, chyba, że kopiuje się znaczące ilości kodu. Na przykład, stworzenie programu, który będzie wykorzystywać kilka fragmentów kodów z tej książki, nie wymaga uzyskania zezwolenia. Sprzedaż lub dystrybucja płyt CD-ROM, które zawierają przykłady pochodzące z książki, wymaga już uzyskania zezwolenia. Odpowiedź na zadane publicznie pytanie poprzez przytoczenie nazwy tej książki i podanie odpowiedniego fragmentu kodu nie wymaga zezwolenia. Włączenie znaczącej ilości przykładowych kodów z tej książki do tworzonej dokumentacji danego produktu będzie już wymagało uzyskania zezwolenia.

Będziemy wdzięczni za powoływanie się na tę książkę. Powoływanie na książkę polega zazwyczaj na podaniu jej autora, tytułu, nazwy wydawnictwa oraz numeru ISBN. Na przykład: Robert Love, *Linux. Programowanie systemowe*, Helion 2008, 83-246-1497-4.

Jeśli wydaje się Państwu, że użycie przykładów kodów przekracza zakres dozwolony w tej książce, prosimy o kontakt: *permissions@oreilly.com*.

Podziękowania

Wiele serc i umysłów przyczyniło się do ukończenia tego rękopisu. Chciałbym szczerze podziękować za współpracę i przyjaźń tym osobom, które przez cały czas pomagały mi poprzez słowa zachęty, swoją wiedzę oraz udzielane wsparcie.

Andy Oram jest fenomenalnym wydawcą i człowiekiem. Ten wysiłek nie byłby możliwy bez jego ciężkiej pracy. Andy jest rzadkim przykładem człowieka, który łączy głęboką wiedzę techniczną z poetycką znajomością języka angielskiego.

Brian Jepson przez pewien czas znakomicie pomagał mi jako wydawca, a efekty jego wsparcia są również zauważalne w tej książce.

Książka ta miała szczęście być czytaną i poprawianą przez wyjątkowych recenzentów technicznych, prawdziwych mistrzów swojej sztuki, bez udziału których nie znaczyłaby wiele w porównaniu z jej ostateczną wersją. Recenzentami technicznymi byli: Robert Day, Jim Lieb, Chris Rivera, Joey Shaw oraz Alain Williams. Mimo ich ciężkiej pracy, wszystkie błędy, które pozostały, są moje.

Rachel Head pracowała wspaniale jako redaktor. W wyniku tego mój rękopis został ozdobiony czerwonym atramentem — czytelnicy na pewno docenią jej korekty.

Na podziękowanie zasługują następujące osoby: Paul Amici, Mikey Babbitt, Keith Barbag, Jacob Berkman, Dave Camp, Chris DiBona, Larry Ewing, Nat Friedman, Albert Gator, Dustin Hall, Joyce Hawkins, Miguel de Icaza, Jimmy Krehl, Greg Kroah-Hartman, Doris Love, Jonathan Love, Linda Love, Tim O'Reilly, Aaron Matthews, John McCain, Randy O'Dowd, Salvatore Ribaudo wraz z rodziną, Chris Rivera, Joey Shaw, Sarah Stewart, Peter Teichman, Linus Torvalds, Jon Trowbridge, Jeremy Van-Doren wraz z rodziną, Luis Villa, Steve Weisberg wraz z rodziną, a także Helen Whisnant.

Podziękowania należą się również moim rodzicom — Bobowi i Elaine.

Robert Love
Boston

ROZDZIAŁ 1.

Wprowadzenie — podstawowe pojęcia

W niniejszej książce omówione zostanie programowanie systemowe (ang. *system programming*), które jest sztuką tworzenia oprogramowania systemowego (ang. *system software*). Oprogramowanie systemowe funkcjonuje w warstwie niskopoziomowej, współdziałając bezpośrednio z jądrem oraz głównymi bibliotekami systemowymi. Składa się ono z powłoki systemowej (ang. *shell*), edytora tekstu, kompilatora oraz debuggera, podstawowych narzędzi i demonów systemowych. Elementy te tworzą kompletne oprogramowanie systemowe, oparte na jądrze i bibliotekach języka C. Wiele innych programów (takich jak wysokopoziomowe aplikacje GUI) działa w większości na bardziej abstrakcyjnych poziomach, sięgając w niższe warstwy okazjonalnie lub wcale. Niektórzy programiści poświęcają cały swój czas na tworzenie oprogramowania systemowego; dla innych jest to tylko jednym z wielu zadań. Nie ma jednak programisty, który nie odniósłby korzyści ze zrozumienia zasad programowania systemowego. Bez względu na to, czy programowanie systemowe jest dla niego „racją bytu", czy też zaledwie fundamentem dla bardziej abstrakcyjnych pojęć, stanowi podstawę całego tworzonego oprogramowania.

Książka ta dotyczy programowania systemowego w Linuksie. Linux jest nowoczesnym uniksopodobnym systemem operacyjnym, napisanym od podstaw przez Linusa Torvaldsa oraz luźno powiązaną wspólnotę hakerów z całego świata. Choć Linux dzieli z Uniksem cele i ideologię, jednak nim nie jest. Linux podąża swoją drogą, która w razie potrzeby odbiega od głównego kierunku, i powraca na nią tylko wtedy, gdy zachodzi taka konieczność. Ogólne zasady programowania systemowego w Linuksie są takie same, jak w innych systemach uniksowych. Linux różni się jednak znacząco od tradycyjnych systemów uniksowych — jest wyposażony w wiele dodatkowych funkcji systemowych, ma nowe możliwości, charakteryzuje się odmiennym zachowaniem.

Programowanie systemowe

Całe programowanie w Uniksie jest programowaniem na poziomie systemowym. Od samego początku systemy uniksowe nie posiadały zbyt wielu abstrakcyjnych rozszerzeń wysokopoziomowych. Nawet programowanie w takim środowisku projektowania jak X Window udostępniało w pełnym zakresie główne API (ang. *Application Programming Interface*) systemowe dla

Uniksa. Zgodnie z tym można powiedzieć, że książka ta przedstawia ogólne zasady programowania w Linuksie. Nie zawiera opisów dotyczących otoczenia programistycznego dla Linuksa — nie znajdzie się tu żaden samouczek o używaniu programu *make*. Poruszone natomiast zostaną tematy omawiające dostępne na nowoczesnej maszynie linuksowej API dla programowania systemowego.

Programowanie systemowe jest najczęściej przeciwstawiane programowaniu aplikacji. Programowanie na poziomie systemowym oraz programowanie na poziomie aplikacji różnią się między sobą w pewnych aspektach. Podczas programowania systemowego programiści muszą zwrócić szczególną uwagę na sprzęt oraz system operacyjny, w którym pracują. Oczywiście istnieją też różnice w używanych bibliotekach oraz funkcjach. W zależności od „poziomu", na którym znajduje się tworzona aplikacja, programowanie systemowe może nie być faktycznie zamienne z programowaniem aplikacji, ale przejście z programowania aplikacji do programowania systemowego (i na odwrót) nie jest zadaniem trudnym. Nawet w przypadku, gdy aplikacja znajduje się na samej górze, daleko od najniższych poziomów systemu, wiedza na temat programowania systemowego ma znaczenie. Takie dobre wzorce i style kodowania stosuje się również we wszystkich rodzajach programowania.

Przez ostatnie lata wyodrębniła się pewna tendencja w programowaniu aplikacji. Polega ona na oddalaniu się od poziomu programowania systemowego i zmierzaniu do projektowania wysokopoziomowego poprzez używanie oprogramowania sieciowego (takiego jak JavaScript czy PHP) lub przez wykorzystanie kodu zarządzanego (obecnego w takich językach jak C# lub Java). Ten sposób projektowania nie przepowiada jednak końca programowania systemowego. Ktoś musi napisać interpreter języka JavaScript czy biblioteki wykonawcze dla C#, co samo w sobie jest przecież programowaniem systemowym. Poza tym, projektanci tworzący oprogramowanie w językach PHP lub Java mogą ciągle korzystać z wiedzy na temat programowania systemowego, ponieważ zrozumienie podstawowych mechanizmów wewnętrznych rządzących systemem umożliwia tworzenie lepszego kodu bez względu na to, na jakim poziomie aplikacji kod ten ma się znajdować.

Pomimo wspomnianej tendencji dotyczącej programowania aplikacji, większość programów w Uniksie i Linuksie jest ciągle tworzona na poziomie systemowym. Wiele linii kodu napisano w języku C; kod ten dostępny jest przede wszystkim poprzez interfejsy udostępnione przez bibliotekę języka C oraz jądro. Oto przykłady programów, które powstały przy pomocy tradycyjnego programowania systemowego: Apache, *bash*, *cp*, Emacs, *init*, *gcc*, *gdb*, *glibc*, *ls*, *mv*, *vim* oraz X. Aplikacje te nie znikną szybko z systemu.

Pojęcie programowania systemowego dotyczy często projektowania jądra lub przynajmniej tworzenia sterowników urządzeń. W tej książce, podobnie jak w większości opracowań dotyczących programowania systemowego, nie rozważa się tematów związanych z projektowaniem jądra, natomiast poddaje się szczegółowej analizie programowanie systemowe dotyczące poziomu użytkownika, to znaczy wszystkiego tego, co znajduje się powyżej jądra systemu (wiedza o wewnętrznych mechanizmach jądra jest użytecznym dodatkiem do tego tekstu). Również programowanie sieciowe, czyli zagadnienia dotyczące gniazd i tym podobnych tematów, nie jest omówione w tej książce. Tworzenie sterowników urządzeń oraz programowanie sieciowe to obszerne zagadnienia, które najlepiej objaśniane są w publikacjach poświęconych tym tematom.

Co to jest interfejs poziomu systemowego i jak tworzy się aplikacje poziomu systemowego w Linuksie? Co dokładnie udostępniają biblioteki języka C oraz jądro systemu? Jak należy tworzyć zoptymalizowany kod i jakie możliwości ma Linux? Jakie poprawne funkcje syste-

mowe udostępnia Linux w porównaniu z innymi wariantami systemów uniksowych? Jak to wszystko działa? Odpowiedzi na te pytania można znaleźć w tej książce.

Kamieniami węgielnymi dla programowania systemowego w Linuksie są: funkcje systemowe, biblioteka języka C oraz kompilator języka C. Każdy z nich zasługuje na parę zdań wprowadzenia.

Funkcje systemowe

Programowanie systemowe rozpoczyna się poprzez użycie funkcji systemowych (ang. *system calls*). Funkcje systemowe (w języku angielskim często nazywane skrótowo *syscalls*) to wywołania wykonywane z przestrzeni użytkownika (edytora tekstu, ulubionej gry itd.) do jądra (kluczowego zbioru wewnętrznych funkcji systemu), w celu zażądania pewnej usługi lub zasobu z systemu operacyjnego. Funkcje te mogą być różne, poczynając od znanych `read()` i `write()`, a kończąc na bardziej „egzotycznych", na przykład `get_thread_area()` i `set_tid_address()`.

Linux implementuje o wiele mniej funkcji systemowych niż większość systemów operacyjnych. Na przykład liczba funkcji systemowych dla architektury i386 wynosi około 300 w porównaniu z przypuszczalnie tysiącami dla Microsoft Windows. W jądrze Linuksa każda architektura maszyny (taka jak Alpha, i386 lub PowerPC) implementuje swoją własną listę możliwych funkcji systemowych. Dlatego też funkcje systemowe dostępne w jednej architekturze mogą różnić się od tych, które są udostępnione w innej. Pomimo tego bardzo duży podzbiór funkcji systemowych — więcej niż 90% — zaimplementowany jest we wszystkich architekturach. W niniejszej książce omówiony zostanie współdzielony podzbiór, zawierający wspólne interfejsy.

Wywoływanie funkcji systemowych

Nie jest możliwe bezpośrednie łączenie aplikacji z przestrzeni użytkownika z przestrzenią jądra. Ze względu na bezpieczeństwo i niezawodność aplikacje z przestrzeni użytkownika nie posiadają uprawnień, by bezpośrednio wykonywać kod jądra lub zmieniać jego dane. Jądro musi udostępniać mechanizm, poprzez który aplikacja z poziomu użytkownika może „zasygnalizować", że chce wywołać jakąś funkcję systemową. Aplikacja może „wpaść" (ang. *trap*) do jądra i korzystając z dobrze zdefiniowanego mechanizmu wykonać tylko taki fragment kodu, do którego uzyska dostęp. Szczegóły tego mechanizmu zależą od architektury. W architekturze i386 aplikacja z poziomu użytkownika wykonuje instrukcję przerwania programowego `int` z wartością `0x80`. Instrukcja ta powoduje wejście aplikacji do chronionego „królestwa" — przestrzeni jądra, w której następuje wywołanie programu obsługi (ang. *handler*) przerwania programowego. A co jest procedurą obsługi dla przerwania `0x80`? Program obsługi funkcji systemowej!

Aplikacja informuje jądro poprzez *rejestry maszynowe*, jaką funkcję systemową i z jakimi parametrami chce wykonać. Funkcje systemowe są oznaczane liczbami, poczynając od zera. Aby przeprowadzić poprawne wywołanie funkcji systemowej nr 5 w architekturze i386 (co odpowiada funkcji o symbolicznej nazwie `open()`), aplikacja z poziomu użytkownika musi wstawić liczbę 5 do rejestru `eax`, zanim wykona instrukcję `int`.

Przekazywanie parametrów obsługiwane jest w podobny sposób. W architekturze i386 używa się rejestrów dla każdego możliwego parametru — rejestry `ebx`, `ecx`, `edx`, `esi` oraz `edi` zawierają kolejno pięć pierwszych parametrów. Rzadko występuje funkcja systemowa, która posiada więcej niż pięć parametrów, jeśli jednak się pojawi, pojedynczy rejestr wskazuje na obszar pamięci (bufor) w przestrzeni użytkownika, w którym przechowywane są wszystkie parametry. Oczywiście większość funkcji systemowych posiada tylko kilka parametrów.

W pozostałych architekturach obsługa wywołań funkcji systemowych przebiega w inny sposób, choć sama idea jest taka sama. Programista systemowy zazwyczaj nie wie, w jaki sposób jądro obsługuje wywołania funkcji systemowych. Informacja ta zapisana jest w standardowych konwencjach wywołań funkcji w danej architekturze, które są automatycznie obsługiwane przez kompilator i bibliotekę języka C.

Biblioteka języka C

Biblioteka języka C (*libc*) jest podstawą aplikacji uniksowych. Nawet w przypadku tworzenia programów w innym języku, biblioteka języka C dostępna jest poprzez interfejs programowy z bibliotek wyższego poziomu — dostarcza niezbędnych usług podstawowych oraz ułatwia wywołanie funkcji systemowych. W nowoczesnych systemach linuksowych biblioteka języka C udostępniana jest w pakiecie o nazwie *GNU libc*, w skrócie *glibc*.

Biblioteka GNU C oferuje więcej możliwości, niż sugeruje jej nazwa. Jako uzupełnienie implementacji standardowej biblioteki języka C, *glibc* udostępnia także interfejsy programowe (ang. *wrappers*) dla funkcji systemowych, wsparcie dla wielowątkowości oraz podstawowe udogodnienia dla aplikacji.

Kompilator języka C

Standardowy kompilator języka C dla Linuksa udostępniany jest w pakiecie o nazwie *GNU Compiler Collection* (*gcc*). Początkowo *gcc* był wersją GNU kompilatora języka C (*cc*), a pełna nazwa pakietu brzmiała *GNU C Compiler*. Z biegiem czasu dodano wsparcie dla większej liczby języków programowania. Dlatego też obecnie nazwa *gcc* używana jest w celu opisania ogólnej rodziny kompilatorów GNU. Jednakże *gcc* oznacza również nazwę pliku binarnego, używanego w celu wywołania kompilatora języka C. W niniejszej książce wszelkie odwołania do nazwy *gcc* oznaczają domyślnie program (plik binarny) *gcc*, chyba że z kontekstu wynika co innego.

Kompilator używany w Uniksie (a także Linuksie) jest ściśle związany z programowaniem systemowym, ponieważ pomaga w zaimplementowaniu standardu języka C (opisanego w podrozdziale Standardy języka C) oraz systemowych ABI (opisanych w podrozdziale API i ABI), o czym będzie mowa w dalszej części rozdziału.

API i ABI

Programiści potrzebują zapewnienia, że ich programy będą działać poprawnie teraz i w przyszłości we wszystkich systemach, dla których obiecano wsparcie. Chcą zabezpieczyć programy tak, aby działały poprawnie nie tylko w aktualnej dystrybucji Linuksa, ale także w innych, jak również w pozostałych wspieranych architekturach linuksowych oraz nowszych (lub starszych) wersjach systemu operacyjnego.

Na poziomie systemowym istnieją dwa oddzielne zbiory definicji i opisów, które wpływają na kompatybilność. Jeden z nich to *interfejs programowania aplikacji* (ang. *Application Programming Interface* — API), natomiast drugi to *interfejs binarny aplikacji* (ang. *Application Binary Interface* — ABI). Zbiory te definiują i opisują interfejsy istniejące pomiędzy różnymi elementami oprogramowania komputerowego.

API

API definiuje interfejsy, przy pomocy których jeden element oprogramowania komunikuje się z drugim elementem na poziomie źródłowym. API zapewnia poziom abstrakcji poprzez dostarczanie standardowego zbioru interfejsów (zazwyczaj będących funkcjami), które mogą być wywoływane z jednego elementu oprogramowania (najczęściej niskopoziomowego) poprzez drugi element oprogramowania (zwykle, choć niekoniecznie, będący elementem wysokopoziomowym). Na przykład API mógłby stworzyć abstrakcję pojęcia wyświetlania tekstu na ekranie poprzez udostępnienie grupy funkcji, które dostarczałyby wszystko, co byłoby niezbędne w celu jego wyświetlenia. API jedynie definiuje interfejs; element oprogramowania, który dostarcza rzeczywistego API, nazywany jest *implementacją* API.

API nazywany jest potocznie „umową". Z formalnego punktu widzenia nie jest to poprawna nazwa, ponieważ API nie jest umową dwustronną. Użytkownik API (zazwyczaj oprogramowanie wyższego poziomu) nie ma w ogóle dostępu do jego szczegółów implementacyjnych. Wolno mu używać API wyłącznie w zakresie, w jakim zostało mu to udostępnione lub nie używać go wcale (zgodnie z zasadą „Weź lub zostaw!"). API służy jedynie temu, aby zapewnić *kompatybilność źródłową* w przypadku, gdy oba elementy oprogramowania stosują zdefiniowane interfejsy. Oznacza to, że użytkownik API poprawnie skompiluje swój kod, w którym użyto implementacji API.

Przykładem wziętym z realnego świata jest API zdefiniowany przez standard języka C i zaimplementowany w standardowej bibliotece języka C. To API definiuje rodzinę podstawowych i niezbędnych funkcji, na przykład podprogramów przetwarzających łańcuchy znakowe.

W niniejszej książce wykorzystane zostaną różne istniejące API, takie jak standardowa biblioteka wejścia i wyjścia, opisana w rozdziale 3. Najważniejsze API dla programowania systemowego w Linuksie rozważone zostaną w podrozdziale Standardy.

ABI

Podczas gdy API definiuje interfejs źródłowy, ABI opisuje niskopoziomowy interfejs binarny pomiędzy dwoma lub więcej elementami oprogramowania, istniejący w określonej architekturze. ABI definiuje, w jaki sposób aplikacja może współpracować ze sobą, z jądrem oraz z bibliotekami. ABI zapewnia *kompatybilność binarną*, która gwarantuje, że dany fragment kodu obiektu będzie działać poprawnie, w dowolnym systemie posiadającym taki sam ABI, bez potrzeby ponownej kompilacji programu.

ABI koncentruje się na następujących tematach: konwencje wywołań funkcji, porządek bajtów, użycie rejestrów, wywołanie funkcji systemowych, linkowanie, sposób działania bibliotek oraz format obiektów binarnych. Na przykład konwencja wywołań ustala sposób, w jaki funkcja powinna być wywoływana, jakie argumenty są przekazywane do funkcji, jakie rejestry są chronione, a jakie używane, w jaki sposób proces wywołujący odczytuje zwracany wynik.

Choć wiele razy próbowano zdefiniować pojedyncze ABI dla określonej architektury, używanej przez różne systemy operacyjne (szczególnie dla architektury i386 i systemów Unix), podjęte wysiłki nie zakończyły się sukcesem. Systemy operacyjne (także Linux) mają skłonność do definiowania swoich własnych ABI, które są ze sobą kompatybilne. ABI jest ściśle związany z architekturą; większa część ABI posługuje się pojęciami specyficznymi dla danej maszyny, takimi jak poszczególne rejestry lub instrukcje asemblera. Tak więc każda architektura danej maszyny

posiada swój własny ABI w Linuksie. Właściwie istnieje tendencja, aby odwoływać się do określonego ABI poprzez nazwę odpowiadającej mu maszyny, np. *alpha* czy *x86-64*.

Choć programiści systemowi powinni zwracać uwagę na ABI, nie muszą zazwyczaj uczyć się go na pamięć. ABI narzucany jest poprzez określony *zestaw narzędzi* (ang. *toolchain*) — kompilator, linker itd., poza tym nie jest widoczny. Posiadana wiedza na temat ABI może jednak wspomóc w dążeniu do bardziej optymalnego programowania i jest wymagana, gdy programuje się w kodzie maszynowym lub modyfikuje wspomniany wcześniej zestaw narzędzi (należy to również do programowania systemowego).

ABI dla określonej architektury systemu Linux dostępny jest w Internecie i został zaimplementowany w zestawie narzędzi oraz jądrze.

Standardy

Programowanie systemowe w Uniksie istnieje od dawna. Podstawy programowania uniksowego pozostają niezmienione przez dziesięciolecia, mimo to systemy uniksowe zmieniają się dynamicznie — dodawane są nowe funkcje. Aby przywrócić porządek w powstającym chaosie, grupy standaryzujące kodyfikują zbiory interfejsów systemowych w oficjalne standardy. Istnieje wiele takich standardów, ale Linux oficjalnie nie jest kompatybilny z żadnym z nich. Linux *dąży* do zgodności z dwoma najważniejszymi i dominującymi standardami, jakimi są POSIX i Single UNIX Specification (SUS).

Standardy POSIX oraz SUS dokumentują między innymi API języka C dla interfejsu uniksopodobnego systemu operacyjnego. Faktycznie jednak definiują programowanie systemowe (lub przynajmniej jakiś wspólny podzbiór) dla zgodnych z nimi systemów uniksowych.

Historia POSIX oraz SUS

W połowie lat 80. XX wieku, Instytut Inżynierów Elektryków i Elektroników (Institute of Electrical and Electronics Engineers — IEEE) przewodził pracom zmierzającym do ustandaryzowania systemowych interfejsów uniksowych. Richard Stallman, założyciel Fundacji Wolnego Oprogramowania (Free Software Foundation), zaproponował, aby standard otrzymał nazwę *POSIX*, co jest skrótowcem od *Portable Operating System Interface* („przenośny interfejs systemu operacyjnego").

Pierwszym rezultatem podjętych wysiłków był dokument zatytułowany *IEEE Std 1003.1-1988* (w skrócie: *POSIX 1988)*, opublikowany w roku 1988. W 1990 IEEE uaktualnił standard POSIX, publikując dokument *IEEE Std 1003.1-1990* (*POSIX 1990*). Opcjonalne wsparcie systemów czasu rzeczywistego oraz wielowątkowości udokumentowano odpowiednio w *IEEE Std 1003.1b-1993* (*POSIX 1993* albo *POSIX.1b*) oraz *IEEE Std 1003.1c-1995* (*POSIX 1995* lub *POSIX.1c*). W roku 2001 połączono opcjonalne standardy z podstawowym dokumentem *POSIX 1990*, tworząc jeden standard: *IEEE Std 1003.1-2001* (*POSIX 2001*). Ostatnią poprawkę, opublikowaną w kwietniu 2004 roku, zatytułowano *IEEE Std 1003.1-2004*. Wszystkie główne standardy POSIX są skrótowo nazywane POSIX.1, przy czym poprawka z roku 2004 jest najnowsza.

Pod koniec lat 80. i na początku 90. XX wieku producenci systemów uniksowych zaangażowali się w tak zwaną „wojnę uniksową", w której każdy z nich próbował udowadniać, że produko-

wany przez niego wariant systemu uniksowego jest jedynie słuszny i zasługuje na nazwę Unix. Kilku głównych producentów zjednoczyło się wokół *The Open Group* — konsorcjum przemysłowego utworzonego poprzez połączenie się *Open Software Foundation* (*OSF*) oraz *X/Open*. The Open Group zajmuje się wydawaniem atestów, dokumentów technicznych oraz wykonywaniem testów zgodności. We wczesnych latach 90. XX wieku, gdy trwała „wojna uniksowa", The Open Group opublikowała standard Single Unix Specification. Darmowy SUS szybko stał się popularny, podczas gdy standard POSIX pociągał za sobą spore koszty finansowe. Obecnie SUS zawiera najnowszą wersję standardu POSIX.

Pierwszy SUS opublikowano w 1994 roku. Systemy kompatybilne z SUSv1 mogą otrzymać certyfikat UNIX 95. Drugi SUS został opublikowany w 1997 roku, a systemy z nim kompatybilne posiadają certyfikat UNIX 98. Trzeci i zarazem ostatni SUS — SUSv3 — opublikowano w 2002 roku. Systemy z nim kompatybilne otrzymują certyfikat UNIX 03. SUSv3 poprawia dokument *IEEE Std 1003.1-2001* i integruje go z kilkoma innymi standardami. Gdy funkcje systemowe oraz inne interfejsy opisane w tej książce będą podlegać standardowi POSIX, zostanie to zaznaczone. W przypadku, gdy standardem będzie SUS, nie zostanie to zaznaczone *explicite*, gdyż późniejsza wersja standardu zawiera w sobie wersję wcześniejszą.

Standardy języka C

Słynna książka Dennisa Ritchiego i Briana Kernighana *The C Programming Language* (wydana przez Prentice Hall) od momentu publikacji w 1978 roku przez wiele lat pełniła rolę nieformalnej specyfikacji języka C. Ten standard języka C przeszedł do historii pod nazwą *K&R C*. C szybko wypierał BASIC i inne języki, stając się uniwersalnym językiem programowania dla mikrokomputerów. Dlatego też, aby ustandaryzować ten wówczas całkiem popularny język, Amerykański Narodowy Instytut Standardów (American National Standards Institute — ANSI) utworzył w roku 1983 komitet w celu zaprojektowania oficjalnej wersji języka C. Miało to nastąpić poprzez połączenie elementów i ulepszeń pochodzących od różnych producentów oraz z nowego języka C++. Proces ten trwał długo i był pracochłonny, lecz w 1989 roku stworzono standard *ANSI C*. W 1990 Międzynarodowa Organizacja Normalizacyjna (International Organization for Standarization — ISO) ratyfikowała standard *ISO C90*, opierając się na ANSI C i wprowadzając niewielkie, lecz przydatne modyfikacje.

W 1995 roku ISO opublikowała i uaktualniła (w niewielkim stopniu zaimplementowała) kolejną wersję języka C, zwaną *ISO C95*. Znaczące uaktualnienie języka nastąpiło w 1999 roku i zostało nazwane *ISO C99*. Wprowadzono wiele nowych elementów, między innymi funkcje inline, nowe typy danych, łańcuchy znakowe o zmiennej długości, komentarze w stylu języka C++ oraz nowe funkcje biblioteczne.

Linux i standardy

Jak już wcześniej wspomniano, Linux dąży do kompatybilności ze standardami POSIX i SUS. Udostępnia interfejsy udokumentowane w SUSv3 oraz POSIX.1, wliczając w to opcjonalne wsparcie dla systemów czasu rzeczywistego (POSIX.1b) oraz wielowątkowości (POSIX.1c). Linux próbuje również dopasować swoje zachowanie do wymagań POSIX i SUS. Brak zgodności z tymi standardami oznacza defekt oprogramowania. Zakłada się, że Linux jest kompatybilny z POSIX.1 oraz SUSv3, lecz nie posiada do tej pory żadnych oficjalnych certyfikatów

potwierdzających tę kompatybilność (szczególnie jeśli chodzi o poświadczenia dla każdej z wersji Linuksa). Dlatego też nie można stwierdzić, że Linux jest oficjalnie zgodny z POSIX oraz SUS.

Jeśli chodzi o standardy języka, Linux jest na dobrej pozycji. Kompilator C o nazwie *gcc* jest zgodny ze standardem ISO C99. Dodatkowo *gcc* dostarcza wielu własnych rozszerzeń dla języka C. Rozszerzenia te znane są pod wspólną nazwą *GNU C*, a w niniejszej książce zostały udokumentowane w Dodatku A.

Linux nie ma wspaniałej historii dotyczącej „kompatybilności w przód"[1] (ang. *forward compatibility*), choć obecnie zachowuje się pod tym względem znacznie lepiej. Interfejsy udokumentowane poprzez standardy, takie jak standardowa biblioteka języka C, zawsze będą kompatybilne źródłowo. Kompatybilność binarna zapewniana jest co najmniej w zakresie głównej wersji *glibc*. Ponieważ język C objęty jest standardem, *gcc* zawsze poprawnie skompiluje właściwie napisany program w tym języku, choć rozszerzenia specyficzne dla *gcc* mogą być w przyszłości unieważnione i w końcu usunięte w kolejnych wersjach tego oprogramowania. Najważniejsze, że jądro Linuksa zapewnia stabilność funkcji systemowych. Od momentu zaimplementowania w stabilnej wersji jądra linuksowego, funkcje te są trwale osadzone i nie zmieniają się.

Linux Standard Base (LSB) jest jedną z dystrybucji Linuksa, która w dużym stopniu standaryzuje system. LSB jest projektem kilku producentów Linuksa pod auspicjami organizacji *Linux Foundation* (poprzednio zwanej *Free Standards Group*). LSB rozszerza POSIX i SUS oraz dodaje kilka własnych standardów. Jej zamierzeniem jest dostarczenie standardu binarnego, pozwalającego w kompatybilnych systemach na uruchamianie kodu obiektu bez modyfikacji. Większość producentów Linuksa jest zgodna w pewnym stopniu z LSB.

Książka i standardy

W książce tej celowo unika się rozwlekłych rozważań na temat standardów. Często publikacje traktujące o programowaniu systemowym w Uniksie zbyt dużo miejsca poświęcają na rozważania o zachowaniu się interfejsów w różnych standardach, o implementacji pewnych funkcji systemowych w danych standardach i tym podobnych problemach. W niniejszej książce omówione zostanie przede wszystkim programowanie systemowe dla nowoczesnej wersji Linuksa, która zapewniana jest dzięki najnowszemu jądru (2.6), kompilator języka C *gcc* (4.2) oraz biblioteka języka C (2.5).

Fakt, że interfejsy systemowe są stabilne i niezmienne (projektanci jądra linuksowego zwracają szczególną uwagę, aby nigdy nie modyfikować funkcji systemowych jądra), a także zapewniają pewien poziom kompatybilności źródłowej i binarnej, pozwala na zagłębienie się w szczegóły, bez potrzeby zajmowania się zgodnością z innymi systemami uniksowymi oraz różnymi standardami. Dzięki temu można poświęcić więcej czasu na omawianie specyficznych interfejsów Linuksa, które pozostaną ważne i aktualne przez długi czas. W książce wykorzystuje się gruntowną wiedzę na temat Linuksa, dotyczącą szczególnie implementacji i zachowania się takich komponentów jak *gcc* i jądra systemu. Pozwala to na zapoznanie się z wewnętrznymi mechanizmami systemowymi i na udostępnienie wiedzy o najlepszych metodach pracy oraz wskazówkach optymalizacyjnych pochodzących od doświadczonego weterana.

[1] Doświadczeni użytkownicy Linuksa zapewne pamiętają zmiany *a.out* na ELF, *libc5* na *glibc*, modyfikacje *gcc* itd. Na szczęście takie zdarzenia należą już do przeszłości.

Pojęcia dotyczące programowania w Linuksie

W tym podrozdziale zaprezentowany zostanie zwięzły przegląd usług dostarczanych przez system Linux. Wszystkie systemy uniksowe, włączając w to Linuksa, udostępniają obustronne zestawy abstrakcji i interfejsów. W ten sposób można potocznie zdefiniować system Unix. Abstrakcje takie jak plik czy proces, interfejsy do zarządzania potokami i gniazdami itd. są sednem Uniksa.

Prezentacja ta zakłada, że czytelnik zna otoczenie linuksowe, tzn. posiada wiedzę na temat powłoki systemowej, podstawowych komend oraz kompilacji prostego programu w języku C. Nie jest to przegląd Linuksa lub jego otoczenia programistycznego, lecz raczej omówienie elementów składających się na podstawy programowania systemowego w Linuksie.

Pliki i system plików

Plik jest fundamentalną abstrakcją w Linuksie. Linux zaprojektowano zgodnie z zasadą: „wszystko jest plikiem" (choć nie wprowadzono jej tak restrykcyjnie, jak w innych systemach, na przykład w Plan9[2]). Zgodnie z tym, większość interakcji w systemie polega na czytaniu i zapisywaniu do plików, nawet jeśli obiekt brany pod uwagę nie jest podobny do zwykłego pliku.

Aby uzyskać dostęp do pliku, należy go najpierw otworzyć. Plik może zostać otwarty do czytania, zapisu lub obu tych czynności naraz. Do otwartego pliku można odwołać się poprzez unikalny deskryptor, odwzorowujący związane z nim metadane na sam plik właściwy. Wewnątrz jądra Linuksa deskryptor reprezentowany jest przez liczbę całkowitą (typu `int` w języku C), zwaną *deskryptorem pliku* (ang. *file descriptor*), w skrócie *fd*. Deskryptory pliku osiągalne są z przestrzeni użytkownika i używane bezpośrednio przez jego programy, w celu uzyskania dostępu do plików. Duża część programowania systemowego w Linuksie dotyczy otwierania, modyfikacji, zamykania i innych czynności związanych z deskryptorami pliku.

Pliki zwykłe

To, co potocznie nazywane jest „plikami", w Linuksie zwane jest *plikami zwykłymi* (ang. *regular files*). Plik zwykły składa się z bajtów danych i zorganizowany jest w postaci ciągłego łańcucha zwanego strumieniem bajtów. W Linuksie nie istnieje żadna dodatkowa organizacja lub formatowanie danych wewnątrz pliku. Bajty mogą przyjmować dowolne wartości i organizować się w pliku w dowolny sposób. Na poziomie systemowym Linux nie wymusza żadnej struktury w plikach poza strumieniem bajtów. Niektóre systemy operacyjne, na przykład VMS, udostępniają pliki posiadające zaawansowaną strukturę, wspierając koncepcję *rekordów*. Linux tego nie robi.

Dowolny bajt wewnątrz pliku można zapisać lub odczytać. Operacje te rozpoczynają się od określonego bajtu, definiując położenie wewnątrz pliku. Położenie to zwane jest *pozycją w pliku* (ang. *file position* lub *file offset*). Pozycja w pliku jest podstawowym elementem metadanych, które są związane przez jądro z każdym otwartym plikiem. Gdy plik jest otwierany, pozycja w pliku wynosi zero. Kiedy bajty z pliku są czytane lub zapisywane do niego, pozycja w pliku zazwyczaj zwiększa się. Pozycja w pliku może być również ustawiona „ręcznie" na określoną

[2] Plan9 — system operacyjny stworzony przez Bell Labs, nazywany często następcą Uniksa. Zastosowano w nim wiele nowatorskich pomysłów, jak również zasadę „wszystko jest plikiem".

wartość, nawet poza koniec tego pliku. Taka próba zapisu spowoduje, że brakujące bajty, znajdujące się między końcem pliku a zapisywaną pozycją, zostaną dołączone do pliku i wypełnione zerami. Choć możliwe jest zapisywanie danych poza koniec pliku, nie można zapisać bajtów przed jego początkiem. Taka procedura nie ma większego sensu, a jej użyteczność jest znikoma. Pozycja w pliku rozpoczyna się od zera i nie może przyjmować wartości ujemnych. Zapisywanie bajtu wewnątrz pliku nadpisuje element, który się przedtem tam znajdował. Dlatego też nie jest możliwe rozszerzenie pliku przez pisanie wewnątrz niego. Najwięcej zapisów do pliku występuje na jego końcu. Maksymalna wartość pozycji w pliku ograniczona jest wyłącznie rozmiarem typu istniejącego w języku C i przeznaczonego do jego przechowywania — dla współczesnego Linuksa wynosi ona 64 bity.

Rozmiar pliku jest mierzony w bajtach i zwany *długością*. Długość jest po prostu liczbą bajtów w ciągłym łańcuchu tworzącym plik. Może zmieniać się dzięki operacji zwanej *obcinaniem* (ang. *truncation*). Plik może zostać obcięty do nowego rozmiaru — mniejszego niż początkowy, co powoduje usunięcie bajtów z jego końca. Trochę zamieszania wprowadza sama nazwa operacji, ponieważ plik może być również „obcięty" do rozmiaru większego niż początkowy. W tym przypadku nowe bajty (które dodaje się na końcu pliku) wypełniane są zerami. Plik może być pusty (ma długość równą zeru) i wówczas nie zawiera żadnych bajtów. Maksymalna długość pliku, podobnie jak w przypadku maksymalnej wartości pozycji, ograniczona jest jedynie rozmiarem typu istniejącego w języku C i używanego przez jądro Linuksa w celu zarządzania plikami. Określone systemy plików mogą jednak wprowadzać własne ograniczenia, ustalając mniejszą wartość maksymalnej długości.

Pojedynczy plik może być otwarty kilka razy przez ten sam lub inny proces. Każdy otwarty egzemplarz pliku otrzymuje unikalny deskryptor; procesy są w stanie współdzielić swoje deskryptory, dzięki czemu pojedynczy deskryptor może być używany przez więcej niż jeden proces. Jądro nie nakłada żadnych ograniczeń dla jednoczesnego dostępu do pliku. Wiele procesów może w tym samym czasie czytać z tego samego pliku lub zapisywać do niego dane. Rezultaty takiego równoległego dostępu zależą od kolejności pojedynczych operacji i są nieprzewidywalne. Programy z przestrzeni użytkownika powinny standardowo koordynować współpracę między procesami, by zapewnić odpowiednią synchronizację operacji.

Choć pliki są zwykle dostępne pod określonymi nazwami, w rzeczywistości jednak nie są z nimi bezpośrednio związane. Plik jest powiązany z tzw. *i-węzłem* (ang. *inode*, od *information node*, czyli *węzła informacji*), który otrzymuje unikalną wartość liczbową. Nazywana jest ona *numerem i-węzła* (ang. *inode number*, często skracane do *i-number* lub *ino*). I-węzeł przechowuje metadane związane z plikiem, takie jak datę modyfikacji, nazwę właściciela, typ, długość oraz położenie danych — ale nie nazwę pliku! Jest obiektem fizycznym, umieszczonym na dysku w uniksopodobnych systemach plików, a zarazem jednostką pojęciową, reprezentowaną przez strukturę danych jądra Linuksa.

Katalogi i dowiązania

Dostęp do pliku poprzez numer jego i-węzła jest niewygodny (to ukryta luka w zabezpieczeniach), dlatego pliki otwierane są zawsze z przestrzeni użytkownika przy pomocy nazwy, a nie numeru i-węzła. Katalogi używane są w celu udostępniania nazw, dzięki którym możliwe jest używanie plików. Katalog odwzorowuje nazwy przyjazne dla człowieka na numery i-węzłów. Para elementów: nazwa oraz i-węzeł zwana jest *dowiązaniem* (ang. *link*). Fizycznie istniejąca na

dysku forma tego odwzorowania — tablica prosta, rozproszona lub cokolwiek — jest zaimplementowana i zarządzana przez kod jądra, który obsługuje dany system plików. Katalog jest dostępny i widoczny jako zwykły plik, który zawiera wyłącznie odwzorowanie nazw na i-węzły. Jądro bezpośrednio używa tego odwzorowania, by zamienić nazwę na i-węzeł.

Gdy aplikacja z poziomu użytkownika chce otworzyć dany plik, jądro otwiera katalog zawierający pliki i szuka potrzebnej nazwy. Następnie, poprzez nazwę, jądro uzyskuje numer i-węzła, a potem sam i-węzeł. I-węzeł zawiera metadane związane z plikiem, zawierające jego położenie na dysku oraz informacje o danych.

Na początku istnieje tylko jeden katalog na dysku, zwany *katalogiem głównym* (ang. *root directory*). Zazwyczaj reprezentowany jest przez ścieżkę, której nazwa składa się z ukośnika: /. Powszechnie wiadomo, że w systemie istnieje zazwyczaj dużo katalogów. Skąd jądro wie, *który* katalog należy przeszukać, by znaleźć właściwą nazwę pliku?

Jak wspomniano już wcześniej, katalogi są czymś w rodzaju plików zwykłych. Mają one nawet dołączone i-węzły. Zatem dowiązania wewnątrz katalogów mogą wskazywać na i-węzły innych katalogów. Oznacza to, że katalogi mogą zagnieżdżać się wewnątrz innych katalogów, tworząc hierarchię (drzewo katalogów). Pozwala to na używanie *ścieżek* (ang. *pathnames*), do czego wszyscy użytkownicy Uniksa są przyzwyczajeni — na przykład */home/blackbeard/landscaping.txt*.

Gdy jądro musi otworzyć plik określony przez powyższą ścieżkę, następuje przejście po każdym elemencie katalogu (zwanym *dentry* w wewnętrznej terminologii jądra) w celu znalezienia i-węzła kolejnego elementu. Biorąc pod uwagę ostatni przykład, jądro rozpocznie przeszukiwanie od katalogu głównego /, otrzyma następnie i-węzeł dla katalogu *home*, wejdzie do niego, uzyska i-węzeł dla katalogu *blackbeard*, do którego również wejdzie, i ostatecznie otrzyma i-węzeł dla pliku *landscaping.txt*. Operacja ta zwana jest *tłumaczeniem katalogu* lub *tłumaczeniem ścieżki* (ang. *directory resolution* lub *pathname resolution*). Jądro Linuksa wykorzystuje również obszar pamięci, zwany *buforem podręcznym katalogu* (ang. *dentry cache*), w którym przechowuje się wyniki tłumaczeń katalogów, co powoduje szybsze sprawdzenie w przypadku zjawiska czasowej lokalizacji[3].

Ścieżka rozpoczynająca się od katalogu głównego zwana jest *pełną* (ang. *fully qualified*) lub *bezwzględną ścieżką* (ang. *absolute pathname*). Niektóre ścieżki nie są pełne — rozpoczynają się od innego katalogu (np. *todo/plunder*). Ścieżki te zwane są *ścieżkami względnymi* (ang. *relative pathnames*). Jądro rozpoczyna tłumaczenie takiego typu ścieżki i łączy od *aktualnego katalogu roboczego*. Stamtąd następuje wejście do katalogu *todo*, a następnie pobranie i-węzła dla katalogu *plunder*.

Chociaż katalogi traktowane są jak zwykłe pliki, nie mogą jednak być podobnie jak one otwierane ani modyfikowane. Do ich przetworzenia wykorzystuje się specjalny zestaw funkcji systemowych. Te funkcje umożliwiają wykonywanie jedynych sensownych w tym przypadku operacji, jakimi są dodawanie lub usuwanie dowiązań. Gdyby przestrzeń użytkownika miała uprawnienia do modyfikacji katalogów bez pośrednictwa jądra, zbyt łatwo mogłoby dojść do załamania systemu z powodu jednego prostego błędu.

[3] *Czasowa lokalizacja* (ang. *temporal locality*) oznacza wysokie prawdopodobieństwo dostępu do określonego zasobu, po którym następuje inny dostęp do tego samego zasobu. Wiele zasobów w komputerze wykazuje czasową lokalizację.

Dowiązania twarde

Opierając się na informacjach przedstawionych do tej pory, można wysnuć wniosek, że nic nie stoi na przeszkodzie, aby wiele różnych nazw było tłumaczonych na ten sam i-węzeł. Tak jest w rzeczywistości. Gdy wiele dowiązań odwzorowuje różne nazwy na ten sam i-węzeł, mamy do czynienia z *dowiązaniem twardym* (ang. *hard link*).

Dowiązania twarde zezwalają złożonym strukturom systemu plików, posiadającym wiele ścieżek, na wskazywanie tych samych danych. Dowiązania twarde mogą się znajdować w tym samym katalogu lub w różnych katalogach — w obu przypadkach jądro w prosty sposób przetłumaczy ścieżkę na właściwy i-węzeł. Określony i-węzeł, wskazujący na pewien fragment danych, może być dowiązany twardo ze ścieżkami */home/bluebeard/map.txt* oraz */home/blackbeard/treasure.txt*.

Usuwanie pliku uruchamia proces *odłączenia* (ang. *unlinking*) go od struktury katalogu — usuwa się i-węzeł pliku oraz jego nazwę z katalogu. Ponieważ Linux udostępnia dowiązania twarde, podczas każdej operacji odłączania system plików nie może skasować i-węzła wraz ze związanymi z nim danymi. Co stanie się, jeśli jakieś inne dowiązanie twarde istnieje jeszcze w systemie plików? Aby upewnić się, że plik nie będzie usunięty, dopóki *wszystkie* dowiązania do niego nie zostaną zlikwidowane, każdy i-węzeł posiada *licznik dowiązań*, który zawiera liczbę dowiązań istniejących w danym systemie plików. Gdy ścieżka jest odłączana, wartość licznika dowiązań maleje o jeden. I-węzeł oraz związane z nim dane są usuwane z systemu plików tylko wtedy, gdy licznik wskaże zero.

Dowiązania symboliczne

Dowiązania twarde nie mogą obejmować wielu systemów plików, gdyż numer i-węzła ma znaczenie wyłącznie w aktualnym systemie plików. Aby umożliwić dowiązaniom istnienie również w innych systemach plików, a jednocześnie uprościć ich działanie i zmniejszyć przezroczystość, systemy uniksowe implementują także *dowiązania symboliczne* (ang. *symbolic links*, skracane często do *symlinks*).

Dowiązanie symboliczne podobne jest do pliku zwykłego. Posiada swój własny i-węzeł oraz obszar danych zawierający pełną nazwę ścieżki do pliku dowiązanego. Oznacza to, że dowiązanie symboliczne może wskazywać na dowolne miejsce, włączając również pliki i katalogi znajdujące w innych systemach plików, a nawet te nieistniejące. Dowiązanie symboliczne, które wskazuje na nieistniejący plik, nazywane jest *dowiązaniem uszkodzonym* (ang. *broken link*).

Dowiązania symboliczne powodują w systemie większe obciążenie niż dowiązania twarde, gdyż ich tłumaczenie jest w rzeczywistości tłumaczeniem dwóch plików: właściwego dowiązania symbolicznego oraz pliku dowiązanego. Dowiązania twarde nie wnoszą dodatkowego narzutu — nie ma różnicy pomiędzy dostępem do pliku dowiązanego wielokrotnie, a dostępem do pliku dowiązanego tylko raz w danym systemie plików. Obciążenie powodowane przez dowiązanie symboliczne jest minimalne, mimo to wciąż jest uważane za negatywną cechę tego rozwiązania.

Dowiązania symboliczne są również mniej przezroczyste niż całkowicie przezroczyste dowiązania twarde. Gdy mamy do czynienia z tymi ostatnimi, trzeba dołożyć starań, aby odkryć, że plik jest dowiązany więcej niż tylko raz! Z drugiej strony, działania na dowiązaniach symbo-

licznych wymagają specjalnych funkcji systemowych. Ten brak przezroczystości jest często uważany za pozytywną cechę, gdyż dowiązanie symboliczne jest raczej *skrótem* (ang. *shortcut*) niż faktycznym połączeniem wewnątrz systemu plików.

Pliki specjalne

Pliki specjalne (ang. *special files*) są obiektami jądra reprezentowanymi przez pliki. Unix od dawna udostępniał kilka różnych plików specjalnych. Linux dostarcza cztery: pliki urządzeń blokowych, urządzeń znakowych, nazwane potoki oraz gniazda domeny uniksowej. Dzięki plikom specjalnym można umieścić pewne abstrakcje w systemie plików, zgodnie z paradygmatem: „wszystko jest plikiem". Linux dostarcza funkcji systemowej dla tworzenia pliku specjalnego.

Dostęp do urządzeń w Uniksie realizowany jest poprzez pliki urządzeń, które zachowują się i wyglądają jak zwykłe pliki. Pliki urządzeń mogą być otwierane, można z nich czytać lub do nich zapisywać, dzięki czemu z poziomu użytkownika istnieje możliwość dostępu i modyfikacji urządzeń systemowych (występujących fizycznie oraz wirtualnie). Urządzenia uniksowe dzielą się na dwie grupy: *urządzenia znakowe* (ang. *character devices*) oraz *urządzenia blokowe* (ang. *block devices*). Każdy typ urządzenia posiada własny plik specjalny.

Urządzenie znakowe dostępne jest jako ciągła kolejka bajtów. Sterownik urządzenia umieszcza bajty w kolejce, jeden po drugim, a przestrzeń użytkownika odczytuje je w takiej kolejności, w jakiej zostały one w niej umieszczone. Przykładem urządzenia znakowego jest klawiatura. Jeśli użytkownik wpisze na przykład słowo „hak", aplikacja będzie czytać z urządzenia klawiatury kolejne litery *h*, *a* oraz *k*. Jeśli nie ma już więcej znaków do odczytania, urządzenie zwraca znacznik końca pliku (EOF). Pomijanie znaku lub czytanie znaków w innej kolejności nie ma sensu. Urządzenia znakowe dostępne są poprzez *pliki urządzeń znakowych*.

W odróżnieniu od urządzenia znakowego, urządzenie blokowe widoczne jest jako tablica bajtów. Sterownik odwzorowuje bajty na urządzenie o dostępie swobodnym, a przestrzeń użytkownika uzyskuje dostęp do dowolnych elementów z tablicy i może na przykład odczytać bajt 12., następnie bajt 7. i ponownie bajt 12. Urządzenia blokowe zasadniczo służą do przechowywania danych. Dyski twarde, stacje dyskietek, stacje CD-ROM oraz pamięć flash to przykłady urządzeń blokowych. Są one dostępne poprzez *pliki urządzeń blokowych*.

Nazwane potoki (ang. *named pipes*), często zwane *FIFO* (od ang. *first in, first out* — „pierwszy na wejściu, pierwszy na wyjściu"), są mechanizmem *komunikacji międzyprocesowej* (ang. *interprocess communication*, IPC), który udostępnia kanał łączności poprzez deskryptor pliku. Kanał ten dostępny jest poprzez plik specjalny. Zwykłe potoki pozwalają na podłączenie wyjścia jednego programu do wejścia drugiego; są one tworzone w pamięci poprzez wywołanie funkcji systemowej i nie istnieją w systemie plików. Nazwane potoki działają jak zwykłe potoki, lecz dostępne są poprzez plik, zwany *plikiem specjalnym FIFO*. Niezależne procesy mogą podłączyć się do takiego pliku i nawiązać komunikację między sobą.

Gniazda (ang. *sockets*) są ostatnim typem pliku specjalnego. Są one zaawansowaną formą komunikacji międzyprocesowej, która pozwala na połączenie ze sobą dwóch różnych procesów znajdujących się nie tylko w tej samej maszynie. W rzeczywistości gniazda są podstawą programowania sieciowego i internetowego. Istnieje wiele odmian gniazd, na przykład gniazda domeny uniksowej, które stosuje się w komunikacji wewnątrz maszyny lokalnej. Gniazda łączące się przez Internet używają nazwy serwera i pary portów w celu identyfikacji komputera docelowego, natomiast gniazda domeny uniksowej wykorzystują plik specjalny znajdujący się w systemie plików, zwany często plikiem gniazda.

Systemy plików i przestrzenie nazw

Linux, podobnie jak wszystkie systemy uniksowe, dostarcza globalną i ujednoliconą *przestrzeń nazw* (ang. *namespace*) dla plików i katalogów. Pewne systemy operacyjne przeznaczają różne dyski i napędy dla oddzielnych przestrzeni nazw — na przykład, plik na dyskietce może być dostępny poprzez ścieżkę *A:\plank.jpg*, natomiast dysk twardy może być osiągalny pod nazwą *C:*. W Uniksie taki sam plik na dyskietce dostępny jest poprzez ścieżkę */media/floppy/plank.jpg* lub nawet przez */home/captain/stuff/plank.jpg* i znajduje się obok plików pochodzących z innych mediów. To oznacza, że w Uniksie przestrzeń nazw jest ujednolicona.

System plików (ang. *filesystem*) jest zbiorem plików i katalogów. Posiada sformalizowaną i poprawną hierarchię. Systemy plików mogą być indywidualnie dodawane i usuwane z globalnej przestrzeni nazw plików i katalogów. Operacje te zwane są *montowaniem* (ang. *mounting*) i *odmontowaniem* (ang. *unmounting*). Każdy system plików montowany jest w określonym miejscu w przestrzeni nazw, czyli w tzw. *punkcie montowania* (ang. *mounting point*). Główny katalog danego systemu plików jest wówczas dostępny w określonym miejscu montowania. Można zamontować płytę CD w katalogu */media/cdrom*, co spowoduje umieszczenie głównego katalogu tej płyty w podanym katalogu. Pierwszy zamontowany system plików jest umieszczony w katalogu głównym przestrzeni nazw, oznaczanym przez ukośnik /. Nazywany jest *głównym systemem plików* (ang. *root filesystem*), który zawsze występuje w systemach linuksowych. Nie jest natomiast wymagane montowanie innych systemów plików.

Systemy plików zwykle istnieją fizycznie (są przechowywane na dysku). Linux wspiera również *wirtualne systemy plików*, istniejące wyłącznie w pamięci, oraz *sieciowe systemy plików*, które istnieją na maszynach zdalnych. Fizyczne systemy plików znajdują się w blokowych urządzeniach przechowywania danych, na przykład w płytach CD, dyskietkach, kartach pamięci flash czy na twardych dyskach. Niektóre z tych urządzeń są *partycjonowane* (ang. *partitionable*), co oznacza, że mogą być przydzielone wielu różnym systemom plików, z których każdy może być indywidualnie obsługiwany. Linux potrafi pracować z wieloma systemami plików — na pewno z tymi, które zna przeciętny użytkownik, wliczając w to systemy plików ukierunkowane na media (na przykład *ISO9660*), sieciowe (*NFS*), macierzyste (*ext3*), inne uniksowe systemy plików (*XFS*), a także te pochodzące z odmiennych systemów operacyjnych (*FAT*).

Najmniejszą adresowalną jednostką w urządzeniu blokowym jest *sektor*. Rozmiary sektorów określają fizyczny poziom jakości dla danego urządzenia — posiadają różne wartości równe potęgom liczby 2, przy czym najczęściej używana jest wartość 512 bajtów. Urządzenie blokowe nie może przesłać lub podłączyć się do jednostki danych mniejszej niż sektor; wszystkie operacje wejścia i wyjścia są ograniczane wielokrotnością sektorów.

Podobnie, najmniejszą logicznie adresowalną jednostką w systemie plików jest *blok*. Blok jest pojęciem abstrakcyjnym dla systemu plików, a nie dla fizycznego nośnika, na którym jest umieszczony. Jest zwykle wielokrotnością rozmiaru sektora, równą najczęściej wartości potęgi liczby 2. Bloki są zwykle większe niż sektor, lecz muszą być mniejsze od *rozmiaru strony*[4], czyli najmniejszej jednostki adresowalnej przez sprzętowy układ zarządzania pamięcią. Popularne rozmiary bloków to 512 bajtów, 1 kB i 4 kB.

Patrząc z perspektywy historycznej, systemy uniksowe posiadały wyłącznie jedną współdzieloną przestrzeń nazw widoczną dla wszystkich użytkowników i procesów w systemie opera-

[4] *Rozmiar strony* to sztuczne ograniczenie jądra, przyjęte w celu jego uproszczenia. Ograniczenie to może zostać zlikwidowane w przyszłości.

cyjnym. Linux podchodzi do tego tematu w sposób nowatorski i obsługuje *przestrzenie nazw dla procesu* (ang. *per-process namespaces*), pozwalające każdemu z nich opcjonalnie posiadać własny widok plików systemu oraz hierarchii katalogów[5]. Domyślnie każdy proces dziedziczy przestrzeń nazw od swojego rodzica, lecz może też stworzyć swoją przestrzeń nazw z własnym zestawem punktów montowania oraz unikalnym katalogiem głównym.

Procesy

Oprócz plików, które są podstawową abstrakcją w systemie Unix, drugim głównym pojęciem abstrakcyjnym są procesy. Procesy są wykonywanymi kodami obiektów, czyli aktywnymi, żyjącymi i działającymi programami. Proces to jednak coś więcej niż tylko kod obiektu — składa się z danych, zasobów, ze stanu procesu, a także ze zwirtualizowanego komputera.

Proces rozpoczyna swoje „życie" jako wykonywalny kod obiektu, będący uruchamialnym kodem maszynowym zapisanym w odpowiednim formacie, zrozumiałym dla jądra (najbardziej popularny format w Linuksie to ELF). Format wykonywalny zawiera metadane oraz wiele *sekcji* z kodem i danymi. Sekcje są ciągłymi fragmentami kodu obiektu, ładowanymi do ciągłych fragmentów pamięci. Wszystkie bajty w sekcji są generalnie używane w podobnym celu, mają takie same uprawnienia.

Najważniejszymi i najbardziej popularnymi sekcjami są: *sekcja tekstu*, *sekcja danych* oraz *sekcja bss*. Sekcja tekstu zawiera kod wykonywalny oraz dane wyłącznie do odczytu (np. stałe programu) i ma zazwyczaj uprawnienia tylko do odczytu i wykonania. Sekcja danych zawiera zainicjalizowane dane, takie jak zmienne języka C, posiadające zdefiniowane wartości — ma zazwyczaj uprawnienia do zapisu i odczytu. Sekcja bss zawiera niezainicjalizowane zmienne globalne. Ponieważ standard C wymusza, aby zmienne domyślnie zawierały zera, nie ma potrzeby, by przechowywać te zera w kodzie obiektu na dysku. Zamiast tego kod obiektu może po prostu zapisać listę niezainicjalizowanych zmiennych w sekcji bss, a jądro przypisze *stronę zerową* (ang. *zero page*) dla tej sekcji w momencie załadowania jej do pamięci. Sekcja bss została stworzona wyłącznie w tym celu. Jej nazwa jest skrótem od angielskich słów: *block started by symbol* („blok rozpoczęty od symbolu") lub *block storage segment* („segment przechowywania bloku"). Inne popularne sekcje w formacie wykonywalnym ELF to *sekcja absolutna* (ang. *absolute section*), która zawiera symbole nieprzemieszczalne oraz *sekcja niezdefiniowana* (ang. *undefined section*), przechwytująca wszystkie odwołania adresowe do obiektów nieznajdujących się w żadnej z powyższych sekcji.

Proces związany jest również z różnymi zasobami systemu, zarządzanymi przez jądro. Procesy zwykle żądają dostępu do zasobów i modyfikują je wyłącznie poprzez funkcje systemowe. Do zasobów można zaliczyć zegary, sygnały zawieszone, otwarte pliki, połączenia sieciowe, sprzęt i mechanizmy IPC. Zasoby procesu, a także jego dane oraz statystyki przechowywane są wewnątrz jądra w tzw. *deskryptorze procesu* (ang. *process descriptor*).

Proces jest abstrakcją wirtualizacji. Jądro Linuksa, obsługujące wielozadaniowość z wywłaszczeniem oraz pamięć wirtualną, dostarcza procesowi zwirtualizowany procesor oraz zwirtualizowany widok pamięci. Od strony procesu system wygląda tak, jakby był on tylko jemu dedykowany. Mimo że procesów może być więcej, każdy z nich działa tak, jak gdyby był sam w systemie i sprawował nad nim jedyną kontrolę. Jądro w sposób płynny i niewidoczny

[5] Rozwiązanie to pojawiło się po raz pierwszy w systemie operacyjnym Plan9 firmy Bell Labs.

wywłaszcza i uruchamia procesy, dzieląc pomiędzy nie zasoby procesorów systemowych. Procesy nie „zauważają" różnicy w działaniu. W podobny sposób procesowi przydzielona zostaje pojedyncza, ciągła przestrzeń adresowa, tak jakby był jedynym procesem zarządzającym pamięcią w systemie. Dzięki pamięci wirtualnej i stronicowaniu, jądro zezwala na jednoczesne istnienie wielu procesów w systemie, z których każdy działa w oddzielnej przestrzeni adresowej. Jądro zarządza tą wirtualizacją dzięki obsłudze sprzętowej, dostarczanej przez nowoczesne procesory, pozwalającej systemowi operacyjnemu współbieżnie kontrolować stany wielu niezależnych procesów.

Wątki

Każdy proces składa się z jednego lub większej liczby *wątków wykonawczych* (ang. *threads of execution*), zwykle zwanych po prostu *wątkami*. Wątek jest jednostką aktywności wewnątrz procesu — abstrakcją odpowiedzialną za wykonywany kod, przechowującą informację o stanie procesu.

Większość procesów składa się tylko z jednego wątku; takie procesy zwane są *jednowątkowymi* (ang. *single-threaded*). Procesy, które zawierają wiele wątków, nazywane są *wielowątkowymi* (ang. *multithreaded*). Programy uniksowe były jednowątkowe dzięki charakterystycznej dla tego systemu prostocie działania, szybkości tworzenia procesów oraz sprawnym mechanizmom IPC. Wszystko to zmniejszało zapotrzebowanie na tworzenie programów wielowątkowych.

Stos (ang. *stack*) jest jednym z elementów, z których zbudowany jest wątek. Stos przechowuje jego lokalne zmienne, podobnie jak stos procesora w przypadku systemów bezwątkowych. Oprócz tego wątek zawiera stan procesora oraz aktualne położenie w kodzie obiektu (zwykle przechowywane w liczniku rozkazów). Większość pozostałych składników procesu dzielone jest pomiędzy wszystkie wątki.

Jądro Linuksa implementuje wewnętrznie unikalny widok wątków: są one zwykłymi procesami dzielącymi pewne zasoby, z których najważniejszy to przestrzeń adresowa. W przestrzeni użytkownika Linux implementuje wątki zgodnie ze standardem POSIX 1003.1c (znanym jako *pthreads*). Aktualna implementacja wątków w Linuksie, będąca częścią biblioteki *glibc*, nazwana jest *Native POSIX Threading Library* (NPTL).

Hierarchia procesów

Każdy proces identyfikowany jest poprzez unikalną dodatnią liczbę całkowitą, zwaną *identyfikatorem procesu* (ang. *process ID*, w skrócie *PID*). Identyfikator pierwszego procesu jest równy 1.

Procesy w Linuksie tworzą sztywną hierarchię, tzw. *drzewo procesów* (ang. *process tree*). Jego korzeniem jest pierwszy proces, zwany *procesem inicjalizującym* (ang. *init process*), którym jest zazwyczaj program o nazwie *init(8)*. Nowe procesy tworzone są przy pomocy funkcji systemowej `fork()`. Funkcja ta tworzy kopię procesu wywołującego. Proces oryginalny zwany jest *rodzicem* lub *procesem rodzicielskim* (ang. *parent*); nowy proces nazywany jest *potomkiem* bądź *procesem potomnym* (ang. *child*). Każdy proces (poza pierwszym) posiada swój proces rodzicielski. Jeśli proces rodzicielski zostanie zakończony przed procesem potomka, jądro przypisze proces potomny procesowi inicjalizującemu, który stanie się jego nowym rodzicem.

Gdy proces kończy swoje działanie, nie jest natychmiast usuwany z systemu. Jądro przechowuje część procesu w pamięci, by umożliwić procesowi rodzicielskiemu możliwość sprawdzenia statusu procesu po jego zakończeniu. Zachowanie to znane jest pod nazwą *obsłużenia* (ang.

waiting on) procesu zakończonego. Proces potomny może być całkowicie usunięty, gdy tylko proces rodzica go obsłuży, czyli sprawdzi jego status. Proces, który został zakończony, lecz nie został obsłużony, zwany jest *procesem zombie*. Proces inicjalizujący oczekuje na wszystkie swoje procesy potomne, gwarantując, że procesy, które zmieniły swoich rodziców, nie pozostaną na zawsze procesami zombie.

Użytkownicy i grupy

Autoryzacja w Linuksie realizowana jest poprzez *użytkowników* (ang. *users*) i *grupy* (ang. *groups*). Każdemu użytkownikowi systemu przypisana jest unikalna dodatnia liczba całkowita, zwana *identyfikatorem użytkownika* (ang. *user ID*), w skrócie UID. Każdy proces przypisany jest dokładnie jednemu UID, reprezentującemu użytkownika, który uruchomił dany proces. Taki identyfikator zwany jest *rzeczywistym identyfikatorem użytkownika* (ang. *real UID*) danego procesu. Dla jądra Linuksa UID jest pojęciem dotyczącym wyłącznie użytkowników. Sami użytkownicy odwołują się jednak do siebie i do innych użytkowników poprzez *nazwy użytkowników* (ang. *usernames*), a nie wartości liczbowe. Nazwy użytkowników oraz odpowiadające im identyfikatory użytkowników przechowywane są w pliku */etc/passwd*, natomiast procedury biblioteczne odwzorowują nazwy użytkowników na odpowiadające im identyfikatory.

Podczas procesu logowania użytkownik podaje programowi *login(1)* swoją nazwę oraz hasło. Jeśli dane te są poprawne, program *login(1)* uruchamia domyślną powłokę systemową, która jest również ustalona w pliku */etc/passwd* i przypisuje identyfikator użytkownika do procesu powłoki. Procesy potomne dziedziczą identyfikatory po swoich rodzicach.

Identyfikator użytkownika równy 0 związany jest z *administratorem* (ang. *root*). Administrator posiada specjalne uprawnienia i może zrobić w systemie praktycznie wszystko. Na przykład, tylko on może zmienić identyfikator użytkownika dla procesu. Zgodnie z tym program *login(1)* działa z identyfikatorem administratora.

Oprócz rzeczywistego identyfikatora użytkownika, każdy proces posiada również *efektywny identyfikator użytkownika* (ang. *effective UID*), inaczej zwany EUID, *zapisany identyfikator użytkownika* (ang. *saved UID*) oraz *identyfikator użytkownika dla systemu plików* (ang. *filesystem UID*). Podczas gdy rzeczywisty UID zawsze określa użytkownika, który uruchomił dany proces, efektywny UID może zmieniać się pod pewnymi warunkami, by zezwolić procesowi na uruchomienie z prawami innego użytkownika. Zapisany identyfikator użytkownika przechowuje pierwotną wartość efektywnego UID; jest on używany podczas podejmowania decyzji, jakimi EUID dany użytkownik może się posługiwać. Identyfikator użytkownika dla systemu plików (zazwyczaj równy identyfikatorowi EUID) jest używany w celu kontroli dostępu do systemów plików.

Każdy użytkownik może należeć do jednej lub więcej grup, wliczając w to *grupę podstawową* (ang. *primary group*) oraz pewien (być może pusty) zbiór *grup dodatkowych* (ang. *supplemental groups*). Grupy podstawowe zapamiętane są w systemie w pliku */etc/passwd*, natomiast grupy dodatkowe w */etc/group*. Każdy proces jest więc związany z odpowiednim *identyfikatorem grupy* (ang. *group ID*), zwanym w skrócie GID. Podobnie jak w przypadku UID, proces posiada cztery różne identyfikatory grupy: rzeczywisty GID, efektywny GID, zapisany GID oraz GID dla systemu plików. Procesy są raczej związane z grupą podstawową użytkownika, a nie z jakąś grupą dodatkową.

Pewne procedury bezpieczeństwa pozwalają procesom na przeprowadzanie danych operacji tylko w przypadku, gdy procesy te spełniają określone kryteria. Zabezpieczenia te zostały zaimplementowane w sposób bardzo uproszczony: tylko procesy z identyfikatorem użytkownika równym 0 posiadały uprawnienia dostępu. Obecnie Linux zastąpił ten system bezpieczeństwa bardziej ogólnym systemem *uprawnień*. Zamiast zwykłej kontroli binarnej, system uprawnień pozwala jądru na definicję reguł dostępu opartą na bardziej precyzyjnych ustawieniach.

Uprawnienia

Standardowe uprawnienia plików oraz mechanizmy zabezpieczeń są w Linuksie takie same, jak we wcześniejszych wersjach Uniksa.

Każdy plik posiada atrybuty autoryzacji, którymi są: właściciel pliku, grupa oraz zbiór bitów uprawnień. Bity opisują uprawnienia użytkownika, grupy oraz pozostałych osób do odczytu, zapisu i uruchomienia pliku; każdej z tych trzech klas przypisane są po trzy bity — tworzą w sumie wartość dziewięciobitową. Właściciele i uprawnienia przechowywane są w i-węźle pliku.

Dla plików zwykłych uprawnienia są oczywiste. Określają możliwość pliku do zapisu, uruchomienia lub otwarcia do odczytu. Dla plików specjalnych uprawnienia do odczytu i zapisu są takie same, jak dla plików zwykłych. To, co rozumie się poprzez odczyt i zapis, zależy wyłącznie od rodzaju danego pliku specjalnego. Uprawnienia do uruchomienia są ignorowane w plikach specjalnych. Uprawnienia odczytu dla katalogów pozwalają na wyświetlenie ich zawartości, natomiast uprawnienia uruchomienia zezwalają na wejście do danego katalogu oraz użycie jego nazwy w ścieżce. W tabeli 1.1. pokazano wszystkie dziewięć bitów uprawnień, ich wartości zapisane w ósemkowym systemie liczbowym (w taki sposób często zapisuje się dziewięć bitów), odpowiadające im wartości tekstowe (wyświetlane np. przez komendę `ls`) oraz zamieszczono krótki opis.

Tabela 1.1. Bity uprawnień oraz ich wartości

Bit	Wartość ósemkowa	Wartość tekstowa	Uprawnienie
8	400	`r--------`	Właściciel może czytać z pliku
7	200	`-w-------`	Właściciel może zapisywać do pliku
6	100	`--x------`	Właściciel może uruchomić plik
5	040	`---r-----`	Grupa może czytać z pliku
4	020	`----w----`	Grupa może zapisywać do pliku
3	010	`-----x---`	Grupa może uruchomić plik
2	004	`------r--`	Pozostali użytkownicy mogą czytać z pliku
1	002	`-------w-`	Pozostali użytkownicy mogą zapisywać do pliku
0	001	`--------x`	Pozostali użytkownicy mogą uruchomić plik

W ramach dodatku do historycznych uprawnień Uniksa, Linux oferuje również *listy kontroli dostępu* (ang. *access control lists),* zwane w skrócie ACL. ACL pozwalają na ustalanie szczegółowych, dokładnych uprawnień oraz zasad bezpieczeństwa, kosztem zwiększonej złożoności działania oraz zajętości dysku.

Sygnały

Sygnały (ang. *signals*) są mechanizmem implementującym system asynchronicznych powiadomień jednokierunkowych. Sygnał może być wysłany z jądra do procesu lub pomiędzy tymi samymi lub różnymi procesami. Sygnały zwykle informują proces o pewnym zdarzeniu. Przykładami takich zdarzeń mogą być wystąpienie błędu segmentacji lub naciśnięcie przez użytkownika klawiszy *Ctrl+C*.

Jądro Linuksa implementuje około 30 sygnałów (dokładna liczba zależy od architektury). Każdy sygnał reprezentowany jest przez stałą liczbową oraz nazwę tekstową. Na przykład sygnał `SIGHUP`, używany w celu powiadomienia o zawieszeniu działania terminalu, posiada wartość 1 w architekturze i386.

Procesy mogą sterować swoim działaniem po przyjęciu danego sygnału. Wyjątkiem są sygnały: `SIGKILL` —przerywa działanie procesu oraz `SIGSTOP` — zatrzymuje proces. Procesy mogą zaakceptować domyślną akcję, którą może być — w zależności od rodzaju sygnału — zwykłe przerwanie działania procesu, przerwanie działania procesu i dodatkowo utworzenie pliku zrzutu systemowego, zatrzymanie procesu lub po prostu niewykonanie żadnej akcji. Innymi działaniami, które mogą podjąć procesy, jest jawne zignorowanie sygnału lub jego obsłużenie. Sygnały ignorowane są odrzucane w trybie cichym. Sygnały obsługiwane powodują uruchomienie dostarczonej przez użytkownika funkcji obsługi sygnału. Gdy tylko dany sygnał zostaje odebrany, program użytkownika wykonuje tę funkcję, a następnie (po powrocie z niej) kontynuuje swoje działanie od miejsca, w którym go przerwano.

Komunikacja międzyprocesowa

Jednym z najważniejszych zadań systemu operacyjnego jest umożliwienie procesom dokonywania wymiany informacji oraz powiadamiania się wzajemnie o występujących zdarzeniach. Jądro Linuksa implementuje większość dawnych mechanizmów Uniksa dotyczących IPC (włączając również elementy zdefiniowane oraz ustandaryzowane przez System V oraz POSIX), jak również kilka własnych.

Mechanizmami IPC obsługiwanymi przez Linux są: potoki, potoki nazwane, semafory, kolejki komunikatów, pamięć dzielona oraz futeksy.

Pliki nagłówkowe

Programowanie systemowe w Linuksie wykorzystuje zestaw plików nagłówkowych. Jądro i *glibc* dostarczają plików nagłówkowych używanych w programowaniu na poziomie systemowym. Oferowane są zarówno standardowe pliki nagłówkowe języka C (na przykład `<string.h>`), jak również pliki nagłówkowe typowe dla Uniksa (choćby `<unistd.h>`).

Obsługa błędów

Trywialnym jest stwierdzenie, że kontrola i obsługa błędów w systemie operacyjnym zasługują na najwyższą uwagę. W programowaniu systemowym błąd sygnalizowany jest poprzez zwracaną wartość funkcji oraz opisywany przez specjalną zmienną o nazwie `errno`. Biblioteka *glibc* w sposób przejrzysty oferuje wsparcie dla `errno` w funkcjach bibliotecznych oraz systemowych.

Szeroki zakres interfejsów omawianych w tej książce używa tego mechanizmu w celu przekazywania informacji o błędach.

Funkcje informują program wywołujący o błędzie poprzez specjalną wartość kodu powrotu, która zwykle równa jest `-1` (dokładna wartość zależy od funkcji). Wartość ta informuje program wywołujący o wystąpieniu błędu, lecz nie dostarcza żadnych szczegółów na temat powodów jego wystąpienia. Zmienna `errno` opisuje przyczynę wystąpienia błędu.

Jest następująco zdefiniowana w pliku nagłówkowym `<errno.h>`:

```
extern int errno;
```

Jej wartość jest prawidłowa jedynie bezpośrednio po wystąpieniu błędu (zwykle zasygnalizowanego przez zwrócenie kodu powrotu równego `-1`), a programista może ją modyfikować w każdym miejscu programu.

Zmienna `errno` może być bezpośrednio zapisywana lub odczytywana; jest modyfikowalną l-wartością. Wartość zmiennej `errno` odwzorowana jest na tekstowy opis określonego błędu. Dyrektywa `#define` preprocesora również służy odwzorowaniu na wartość numeryczną `errno`. Na przykład dyrektywa `#define` preprocesora przypisuje ciąg znakowy `EACCESS` wartości liczbowej 1, co reprezentuje błąd: „dostęp zabroniony". Tabela 1.2. przedstawia standardowe definicje preprocesora dla wartości zmiennej `errno` oraz odpowiednie opisy błędów.

Biblioteka języka C dostarcza zestawu funkcji, aby zamieniać wartość liczbową `errno` na odpowiedni łańcuch tekstowy. Jest to potrzebne w celu stworzenia raportu o błędach, natomiast dla sprawdzania i obsługi błędów można bezpośrednio używać dyrektywy `#define` preprocesora oraz zmiennej `errno`.

Pierwszą funkcją z tego zestawu jest `perror()`:

```
#include <stdio.h>
void perror (const char *str);
```

Funkcja ta wysyła na *stderr* (standardowy strumień błędów) ciąg znaków reprezentujący aktualny błąd, opisany przez zmienną `errno` i poprzedzony przez łańcuch wskazywany przez parametr `str`. Oba łańcuchy oddziela znak dwukropka. Nazwa funkcji, której wywołanie się nie powiodło, powinna być przekazana w parametrze `str`. Na przykład:

```
if (close (fd) == -1)
   perror ("close");
```

Biblioteka języka C dostarcza również funkcje `strerror()` i `strerror_r()`, których prototypy wyglądają następująco:

```
#include <string.h>
char* strerror (int errnum);
```

oraz:

```
#include <string.h>
int strerror_r (int errnum, char *buf, size_t len);
```

Pierwsza z tych funkcji zwraca wskaźnik do łańcucha opisującego błąd przekazany w parametrze `errnum`. Aplikacji nie wolno modyfikować tego łańcucha, lecz może być on zmieniony przez dalsze wywołania funkcji `perror()` oraz `strerror()`. Z tego powodu funkcja ta nie działa bezpiecznie w środowisku wielowątkowym.

Tabela 1.2. Błędy wraz z ich opisami

Dyrektywa #define preprocesora	Opis
E2BIG	Zbyt długa lista argumentów
EACCESS	Dostęp zabroniony
EAGAIN	Należy ponowić próbę
EBADF	Błędny numer pliku
EBUSY	Urządzenie lub zasób są zajęte
ECHILD	Brak procesów potomnych
EDOM	Błąd zakresu argumentów w funkcji matematycznej
EEXIT	Plik już istnieje
EFAULT	Błędny adres
EFBIG	Plik zbyt duży
EINTR	Funkcja systemowa została przerwana
EINVAL	Błędny argument
EIO	Błąd wejścia i wyjścia
EISDIR	Obiekt jest katalogiem
EMFILE	Za dużo otwartych plików
EMLINK	Za dużo dowiązań
ENFILE	Przepełnienie tablicy plików
ENODEV	Niewłaściwe urządzenie
ENOENT	Błędna nazwa pliku lub katalogu
ENOEXEC	Błędny format pliku wykonywalnego
ENOMEM	Brak pamięci
ENOSPC	Brak wolnego miejsca w urządzeniu
ENOTDIR	Obiekt nie jest katalogiem
ENOTTY	Błędna operacja sterująca wejściem i wyjściem
ENXIO	Niewłaściwe urządzenie lub adres
EPERM	Operacja zabroniona
EPIPE	Przerwany potok
ERANGE	Wynik zbyt duży
EROFS	System plików tylko do odczytu
ESPIPE	Błędna operacja wyszukiwania
ESRCH	Nie ma takiego procesu
ETXTBSY	Plik tekstowy jest zajęty
EXDEV	Nieprawidłowe dowiązanie

Funkcja `strerror_r()`obsługuje wątki w sposób bezpieczny. Funkcja ta wypełnia bufor o długości `len`, który jest wskazywany przez parametr wskaźnikowy `buf`. Wywołanie funkcji `strerror_r()` zwraca zero w przypadku sukcesu, a `-1` podczas niepowodzenia, dodatkowo ustawiając wówczas zmienną `errno`.

W przypadku kilku funkcji, cały zakres zwracanych przez nich wartości jest poprawny. W tej sytuacji zmienna `errno` musi zostać wyzerowana przed wywołaniem funkcji, a po użyciu funkcji powinna zostać sprawdzona (takie funkcje ustawiają `errno` na wartość niezerową tylko w momencie wystąpienia faktycznego błędu). Na przykład:

```
errno = 0;
arg = strtoul (buf, NULL, 0);
if (errno)
   perror ("strtoul");
```

Powszechnym błędem w sprawdzaniu wartości zmiennej `errno` jest zapominanie, że może ją zmodyfikować dowolne wywołanie funkcji bibliotecznej lub systemowej. Na przykład poniższy kod jest błędny:

```
if (fsync (fd) == -1)
{
   fprintf (stderr, "Funkcja fsync wykonała się z błędem!\n");
   if (errno == EIO)
      fprintf (stderr, "Błąd wejścia i wyjścia dla pliku %d!\n", fd);
}
```

Jeśli konieczne jest zachowanie wartości zmiennej `errno` pomiędzy wywołaniami funkcji, należy ją zapamiętać:

```
if (fsync (fd) == -1)
{
   int err = errno;
   fprintf (stderr, "Funkcja fsync wykonała się z błędem: %s\n", strerror (errno));
   if (err == EIO)
   {
      /* jeśli błąd dotyczy operacji wejścia i wyjścia, wyjdź z programu */
      fprintf (stderr, "Błąd wejścia i wyjścia dla pliku %d!\n", fd);
      exit (EXIT_FAILURE);
   }
}
```

Dla programów jednowątkowych dostęp do zmiennej `errno` jest globalny, o czym wspominaliśmy już wcześniej w tym podrozdziale. W przypadku programów wielowątkowych, zmienna ta jest jednak przechowywana osobno dla każdego z wątków i dlatego jej użycie jest bezpieczne.

Początek programowania systemowego

W rozdziale pierwszym przedstawiono podstawy programowania systemowego w Linuksie i omówiono ogólnie system operacyjny z punktu widzenia programisty. W następnym rozdziale analizie poddane zostaną podstawowe operacje plikowe wejścia i wyjścia. Uwzględnione zostaną oczywiście operacje czytania z plików i zapisywania do plików. Ponieważ Linux implementuje wiele interfejsów jako pliki, plikowe operacje wejścia i wyjścia są kluczowymi zagadnieniami, niedotyczącymi tylko i wyłącznie plików zwyczajnych.

Tyle słowem wstępu. Teraz zagłębimy się w programowanie systemowe. Do dzieła!

ROZDZIAŁ 2.

Plikowe operacje wejścia i wyjścia

W tym rozdziale omówione zostaną podstawy czytania z plików i zapisywania do nich. Operacje te są istotą systemu uniksowego. W następnym rozdziale analizie poddane zostaną typowe operacje wejścia i wyjścia, obsługiwane przez standardową bibliotekę języka C, a w rozdziale 4. przedstawione zostaną bardziej zaawansowane i wyspecjalizowane interfejsy wejścia i wyjścia dla plików. Ostatecznie w rozdziale 7. poruszone zostaną zagadnienia dotyczące pracy z plikami i katalogami.

Plik musi zostać wcześniej otwarty, zanim będzie można z niego czytać lub do niego zapisywać. Dla każdego działającego procesu jądro zawiera listę jego otwartych plików. Lista ta zwana jest *tablicą plików* (ang. *file table*). Tablica ta jest indeksowana nieujemnymi liczbami całkowitymi, znanymi jako *deskryptory plików* (skrótowo zwanymi *fds*). Każdy element tej tablicy posiada informacje o otwartym pliku, włącznie ze wskaźnikiem do obszaru pamięci, który zawiera pomocniczą kopię i-węzła oraz metadanych, takich jak pozycja w pliku oraz tryby dostępu. Deskryptory plików używane są zarówno w przestrzeni użytkownika, jak i przestrzeni jądra. Pełnią rolę unikalnych znaczników kontekstu dla każdego istniejącego procesu. Otwarcie pliku powoduje zwrócenie deskryptora pliku, który może być używany w kolejnych operacjach (czytanie, pisanie itd.) jako podstawowy parametr dla odpowiednich funkcji systemowych.

Proces potomny otrzymuje domyślnie kopię tablicy plików od swojego rodzica. Lista otwartych plików, tryby dostępu do nich, aktualne pozycje plików itd. pozostają takie same, lecz zmiana dokonana w jednym procesie — np., gdy proces potomny zamknął jakiś plik — nie wpływa na tablicę plików innego procesu. Jednakże, jak zostanie to przedstawione w rozdziale 5., proces potomny może współdzielić z rodzicem jego tablicę plików (w ten sposób funkcjonują wątki).

Deskryptory plików reprezentowane są przez typ `int` języka C. Brak użycia w tym celu specjalnego typu (na przykład `fd_t`) wydaje się być dziwny, lecz jest to typowe dla Uniksa. Każdy proces w Linuksie może jednocześnie otworzyć jedynie określoną liczbę plików. Deskryptory plików rozpoczynają się od zera i rosną do wartości o 1 mniejszej niż maksymalna, dopuszczalna liczba plików otwartych. Domyślnie ta maksymalna wartość wynosi 1024, lecz może zostać podniesiona do wielkości 1048576. Ponieważ liczby ujemne nie są poprawnymi deskryptorami plików, wartość `-1` jest często używana, by zasygnalizować błąd w funkcji, która normalnie zwraca właściwy deskryptor pliku.

Każdy proces posiada domyślnie otwarte co najmniej trzy deskryptory plików o numerach 0, 1 oraz 2, chyba że zostaną one jawnie zamknięte. Deskryptor pliku 0 jest *standardowym wejściem* (ang. *standard in*, w skrócie *stdin*), deskryptor pliku 1 jest *standardowym wyjściem* (ang. *standard out*, w skrócie *stdout*), zaś deskryptor pliku 2 jest *standardowym wyjściem błędów* (ang. *standard error*, w skrócie *stderr*).

Biblioteka języka C dostarcza odpowiednich definicji preprocesora `STDIN_FILENO`, `STDOUT_FILENO` oraz `STDERR_FILENO`, dzięki czemu nie jest konieczne bezpośrednie używanie powyższych wartości liczbowych.

Na uwagę zasługuje fakt, że deskryptory plików mogą zostać użyte nie tylko w przypadku plików zwykłych. Można je wykorzystać, by uzyskać dostęp do plików urządzeń oraz potoków, katalogów i futeksów, kolejek FIFO, a także gniazd. Zgodnie z filozofią: „wszystko jest plikiem", każdy element zdolny do odczytu lub zapisu może być powiązany z deskryptorem pliku.

Otwieranie plików

Najbardziej podstawową metodą dostępu do pliku jest użycie funkcji systemowych `read()` oraz `write()`. Zanim jednak te funkcje zostaną użyte, należy otworzyć plik za pomocą funkcji `open()` lub `creat()`. Gdy tylko operacje na pliku zakończą się, należy zamknąć go przy użyciu funkcji systemowej `close()`.

Funkcja systemowa open()

Plik może zostać otwarty, a deskryptor pliku zwrócony przy użyciu funkcji systemowej `open()`:

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

int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);
```

Funkcja systemowa `open()` odwzorowuje plik, którego nazwa podana jest w parametrze `name`, na deskryptor pliku, zwracany podczas poprawnego wykonania tej funkcji. Pozycja w pliku ustawiona jest na zero, a sam plik ma zdefiniowany tryb dostępu zgodnie ze znacznikami, przekazanymi w parametrze `flags`.

Znaczniki w funkcji open()

Parametr `flags` musi być równy jednej z następujących wartości: `O_RDONLY`, `O_WRONLY` lub `O_RDWR`. Wartości te wymuszają otwarcie pliku odpowiednio w trybach do odczytu, zapisu lub zarówno do odczytu, jak i zapisu.

Na przykład, następujący kod otwiera plik */home/kidd/madagascar* do odczytu:

```
int fd;

fd = open ("/home/kidd/madagascar", O_RDONLY);
if (fd == -1)
   /* błąd */
```

Plik otwarty jedynie do zapisu *nie może* być odczytywany i na odwrót. Proces, który wywołuje funkcję systemową `open()`, musi posiadać wystarczające uprawnienia, by uzyskać żądany dostęp do pliku.

Wartość parametru `flags` może być łączona za pomocą operatora sumy bitowej z jednym lub kilkoma znacznikami modyfikującymi działanie funkcji otwierania pliku:

`O_APPEND`
: Plik zostanie otwarty w *trybie dopisywania* (ang. *append mode*). Oznacza to, że przed każdą operacją zapisu pozycja w pliku aktualizowana jest w taki sposób, aby wskazywać na jego koniec. Realizowane jest to nawet wtedy, gdy inny proces wykonał swój zapis do pliku już po ostatnim zapisie przeprowadzonym przez proces wymuszający ten tryb, co spowodowało zmianę pozycji w pliku (szczegółowa analiza tego trybu omówiona jest w podrozdziale Tryb dopisywania, znajdującym się w dalszej części tego rozdziału).

`O_ASYNC`
: Nastąpi wygenerowanie sygnału (domyślnie jest to `SIGIO`), gdy określony plik stanie się dostępny do odczytu lub zapisu. Ten znacznik może być użyty jedynie dla terminali oraz gniazd, ale nie dla plików zwykłych.

`O_CREAT`
: Jeśli nazwa `name` oznacza plik nieistniejący, nastąpi jego utworzenie przez jądro. Jeśli plik już istnieje, użycie tego znacznika nie spowoduje żadnych zmian, dopóki nie zastosuje się dodatkowo znacznika `O_EXCL`.

`O_DIRECT`
: Plik zostanie otwarty dla bezpośrednich operacji wejścia i wyjścia (opisanych w podrozdziale Bezpośrednie operacje wejścia i wyjścia w dalszej części tego rozdziału).

`O_DIRECTORY`
: Jeśli parametr `name` nie jest nazwą katalogu, wywołanie funkcji `open()` nie powiedzie się. Znacznik ten używany jest wewnętrznie przez funkcję biblioteczną `opendir()`.

`O_EXCL`
: Gdy znacznik ten użyty zostanie razem ze znacznikiem `O_CREAT`, spowoduje, że wywołanie funkcji `open()` nie powiedzie się, gdy plik określony parametrem `name` istnieje już w systemie. Znacznika tego używa się w celu zabezpieczenia się przed sytuacją współzawodnictwa w momencie tworzenia pliku.

`O_LARGEFILE`
: Użyta zostanie 64-bitowa wartość parametru położenia, dzięki czemu możliwa będzie praca z plikami większymi niż 2 GB. Użycie tego znacznika wymuszane jest w architekturach 64-bitowych.

`O_NOCTTY`
: Jeśli podana nazwa pliku dotyczy urządzenia terminalowego (np. */dev/tty*), urządzenie to nie stanie się kontrolnym terminalem procesu, nawet jeśli do tej pory nie posiadał on żadnego terminala. Znacznik ten jest rzadko używany.

`O_NOFOLLOW`
: Jeśli nazwa pliku jest dowiązaniem symbolicznym, wywołanie funkcji `open()` nie powiedzie się (gdy znacznik ten nie jest użyty, następuje zwykłe przetłumaczenie nazwy i plik docelowy zostaje otwarty). Jeśli inne elementy, składające się na podaną ścieżkę, są dowiązaniami symbolicznymi, funkcja wykona się poprawnie. Na przykład, jeśli parametr `name` równy jest

/etc/ship/plank.txt, funkcja wykona się z błędem, gdy *plank.txt* będzie dowiązaniem symbolicznym. Gdy *etc* lub *ship* będą dowiązaniami symbolicznymi, natomiast *plank.txt* nim nie będzie, funkcja wykona się poprawnie.

`O_NONBLOCK`
: Jeśli to będzie tylko możliwe, plik zostanie otwarty w trybie nieblokującym. Wywołanie funkcji systemowej `open()` ani żadna inna operacja nie spowodują zablokowania procesu (przejdzie on do stanu uśpienia) podczas obsługi wejścia i wyjścia. Zachowanie to może być zdefiniowane wyłącznie dla kolejek FIFO.

`O_SYNC`
: Plik zostanie otwarty dla zsynchronizowanych operacji wejścia i wyjścia. Dopóki dane nie znajdą się fizycznie na dysku, żadna operacja zapisu nie zostanie zakończona. Zwykłe operacje czytania są zawsze synchroniczne, dlatego ten znacznik nie powoduje dla nich żadnych zmian. POSIX dodatkowo definiuje znaczniki o nazwach `O_DSYNC` oraz `O_RSYNC`; w Linuksie są one równoznaczne z `O_SYNC` (więcej na ten temat w podrozdziale Znacznik O_SYNC, znajdującym się w dalszej części tego rozdziału).

`O_TRUNC`
: Jeśli plik istnieje i jest zwykłym plikiem otwieranym w trybie zapisu, jego długość zostanie wyzerowana. Użycie znacznika `O_TRUNC` w przypadku kolejek FIFO nie powoduje żadnych zmian. Użycie z innymi rodzajami plików nie jest zdefiniowane. Zastosowanie znacznika `O_TRUNC` razem z `O_RDONLY` również nie jest zdefiniowane, ponieważ plik wymaga dostępu w trybie zapisu, aby mógł być obcięty.

Przykładowy kod otwiera plik o nazwie */home/teach/pearl* w trybie zapisu. Jeśli plik już istnieje, zostanie obcięty do długości równej zeru. Ponieważ znacznik `O_CREAT` nie został użyty, wykonanie funkcji nie powiedzie się, gdy plik nie będzie istnieć w trakcie jej wywołania:

```
int fd;

fd = open ("/home/teach/pearl", O_WRONLY | O_TRUNC);
if (fd == -1)
   /* błąd */
```

Właściciele nowych plików

Ustalenie, który użytkownik jest właścicielem nowo utworzonego pliku, jest proste: UID właściciela jest równy efektywnemu UID procesu, który ten plik utworzył.

Bardziej skomplikowane jest już ustalenie grupy właścicielskiej. Domyślnie plik należy do grupy, której identyfikator równy jest efektywnemu GID dla procesu tworzącego ten plik. Jest to zachowanie charakterystyczne dla Uniksa System V (będącego modelowym wzorcem dla większości zachowań Linuksa) i jednocześnie standardowy sposób postępowania systemu Linux dla tego przypadku.

Aby skomplikować sytuację, system operacyjny BSD definiuje swój własny sposób postępowania: grupa właścicielska pliku równa jest GID katalogu, w którym ten plik się znajduje. Zachowanie to dostępne jest w Linuksie poprzez użycie określonej opcji podczas montowania systemu plików[1]. To także domyślne zachowanie dla systemu Linux w przypadku, gdy nadrzędny katalog posiada ustawiony bit *set group ID* (*SGID*). Choć większość systemów linuksowych działa

[1] Opcje o nazwie `bsdgroups` lub `sysvgroups`.

w sposób charakterystyczny dla Uniksa System V (w którym nowy plik otrzymuje GID od tworzącego go procesu), możliwość zachowania typowego dla BSD (w którym nowy plik uzyskuje GID od swojego katalogu nadrzędnego) powoduje, że kod, który powinien być poprawnie napisany, musi jawnie ustawiać identyfikator grupy poprzez wywołanie funkcji systemowej `chown()` (opisanej w rozdziale 7.).

Na szczęście troska o grupę właścicielską pliku nie jest częstym zjawiskiem.

Uprawnienia nowych plików

Oba omówione formaty funkcji systemowej `open()` są poprawne. Parametr `mode` jest ignorowany, chyba że zostaje utworzony nowy plik. Parametr ten jest więc wymagany, gdy użyty zostanie znacznik `O_CREAT`. Jeśli znacznik ten zostanie użyty bez parametru `mode`, wynik działania funkcji jest nieprzewidywalny i często dość niebezpieczny, nie należy więc o tym zapominać!

Gdy plik jest tworzony, parametr `mode` zawiera jego wzorzec uprawnień. Parametr ten nie jest sprawdzany podczas wywołania funkcji `open()`, dlatego też można wykonać sprzeczne operacje, na przykład otworzyć plik do zapisu z jednoczesnym przyporządkowaniem mu uprawnień tylko do odczytu.

Parametr `mode` to znany w Uniksie bitowy zestaw uprawnień, taki jak na przykład liczba ósemkowa 0644 (oznaczająca, że właściciel ma uprawnienia do czytania i zapisywania, a wszyscy inni posiadają wyłącznie uprawnienia do czytania). Mówiąc językiem technicznym, POSIX zezwala, aby dokładne wartości zależały od implementacji, przez co umożliwia różnym wersjom systemu Unix definiowanie swoich własnych układów bitów uprawnień. Aby zrównoważyć ten brak przenośności, POSIX wprowadził następujący zestaw stałych, które mogą być ze sobą łączone za pomocą operatora sumy bitowej, tworząc wartość parametru `mode`:

`S_IRWXU`
: Właściciel posiada uprawnienia do zapisu, odczytu i uruchamiania.

`S_IRUSR`
: Właściciel posiada uprawnienia do odczytu.

`S_IWUSR`
: Właściciel posiada uprawnienia do zapisu.

`S_IXUSR`
: Właściciel posiada uprawnienia do uruchamiania.

`S_IRWXG`
: Grupa posiada uprawnienia do zapisu, odczytu i uruchamiania.

`S_IRGRP`
: Grupa posiada uprawnienia do odczytu.

`S_IWGRP`
: Grupa posiada uprawnienia do zapisu.

`S_IXGRP`
: Grupa posiada uprawnienia do uruchamiania.

`S_IRWXO`
: Pozostali użytkownicy posiadają uprawnienia do zapisu, odczytu i uruchamiania.

`S_IROTH`
: Pozostali użytkownicy posiadają uprawnienia do odczytu.

`S_IWOTH`
: Pozostali użytkownicy posiadają uprawnienia do zapisu.

`S_IXOTH`
: Pozostali użytkownicy posiadają uprawnienia do uruchamiania.

Faktyczna wartość bitów uprawnień, zapisana na dysku, ustalona zostaje poprzez wykonanie bitowego iloczynu parametru `mode` z bitową negacją aktualnej *maski uprawnień tworzonych plików* (*umask*). W pewnym sensie bity wartości *umask* są po prostu „wyłączane" w parametrze `mode`, przekazywanym do funkcji `open()`. Dlatego też w przypadku wartości *umask* równej 022 i przykładowego parametru `mode` równego 0666 rezultatem tego działania jest 0644 (0666 & ~022). Programista systemowy zwykle nie bierze pod uwagę wartości *umask* podczas ustawiania uprawnień — istnieje ona tylko po to, aby umożliwić użytkownikowi ograniczenie uprawnień, które mogą być nadawane nowym plikom, przy użyciu programów przez niego uruchamianych.

Jako przykład zaprezentowany zostanie kod, otwierający plik w trybie zapisu. Nazwa pliku przekazana zostaje w parametrze `file`. Jeśli plik nie istnieje, zostanie utworzony i otrzyma uprawnienia 0644 (pomimo że parametr `mode` wynosi 0664) przy założeniu, że aktualna wartość *umask* wynosi 022. Jeśli plik istnieje, zostanie obcięty do długości równej zeru:

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC,
  S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IROTH);
if (fd == -1)
   /* błąd */
```

Funkcja creat()

Użycie konstrukcji `O_WRONLY | O_CREAT | O_TRUNC` jest tak powszechne, że stworzona została funkcja systemowa, umożliwiająca utworzenie pliku w takim trybie:

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

int creat (const char *name, mode_t mode);
```

Tak, w nazwie tej funkcji brakuje litery *e*[2]. Ken Thompson, twórca Uniksa, zażartował pewnego razu, że podczas projektowania Uniksa najbardziej opłakiwał właśnie tę brakującą literę.

Oto typowe wywołanie funkcji `creat()`:

```
int fd;

fd = creat (file, 0644);
if (fd == -1)
   /* błąd */
```

[2] Ponieważ funkcja ta tworzy plik, dlatego też poprawnie powinna nazywać się `create()`, zgodnie z angielskim znaczeniem użytego słowa — *przyp. tłum.*

Wywołanie to jest identyczne z poniższym:

```
int fd;

fd = open (file, O_WRONLY | O_CREAT | O_TRUNC, 0644);
if (fd == -1)
   /* błąd */
```

W większości architektur Linuksa[3] `creat()` jest funkcją systemową, mimo że może zostać w prosty sposób zaimplementowana w przestrzeni użytkownika:

```
int creat (const char *name, int mode)
{
   return open (name, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

To powielenie jest pozostałością z czasów, gdy funkcja `open()` posiadała tylko dwa parametry. Obecnie funkcja systemowa `creat()`zachowana jest ze względu na kompatybilność. Implementacja `creat()` w nowych architekturach oparta jest na bibliotece *glibc*.

Wartości zwracane i kody błędów

Obie funkcje `open()` i `creat()` zwracają deskryptor pliku w przypadku sukcesu. W przypadku błędu zwracają wartość `−1` oraz ustawiają odpowiedni kod błędu (w rozdziale 1. omówiono zmienną `errno` i zaprezentowano możliwe wartości błędów). Obsługa błędu w przypadku otwierania pliku nie jest skomplikowana, ponieważ i tak liczba operacji, które należy przeprowadzić, by unieważnić otwarcie pliku, jest niewielka (lub równa zeru). Typową odpowiedzią programu może być poproszenie użytkownika o podanie innej nazwy pliku lub po prostu przerwanie działania.

Czytanie z pliku przy użyciu funkcji read()

Gdy już wiadomo, w jaki sposób plik powinien być otwierany, można zająć się zagadnieniem czytania z pliku. W następnym podrozdziale omówione zostanie zapisywanie do pliku.

Najbardziej podstawowym (i powszechnie stosowanym) mechanizmem używanym w celu czytania z pliku, jest funkcja systemowa `read()`, zdefiniowana w POSIX.1:

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t len);
```

Podczas każdego wywołania funkcji następuje odczytanie pewnej liczby bajtów, która określona jest w parametrze `len`. Czytanie danych rozpoczyna się od aktualnej pozycji w pliku, którego deskryptor przekazany jest w parametrze `fd`. W przypadku poprawnego wywołania, zwrócona zostaje liczba bajtów zapisanych do bufora, wskazywanego przez parametr `buf`. W przypadku błędu, funkcja zwraca `−1` oraz odpowiednio ustawia zmienną `errno`. Pozycja w pliku zostaje zwiększona o przeczytaną z niego liczbę bajtów. Jeśli obiekt reprezentowany przez parametr `fd`

[3] Wcześniej zostało już wspomniane, że funkcje systemowe definiowane są dla danej architektury. Tak więc architektura i386 posiada funkcję systemową `creat()`, natomiast architektura Alpha jej nie posiada. Funkcja `creat()` może oczywiście istnieć w każdej architekturze, lecz może być zdefiniowana jako funkcja biblioteczna, a nie funkcja systemowa.

nie jest zdolny do wykonywania operacji szukania (np. jest to plik urządzenia znakowego), czytanie rozpoczyna się zawsze od „bieżącej" pozycji.

Podstawowy sposób użycia funkcji jest prosty. Poniższy przykład pokazuje kod czytający bajty do bufora `word` z pliku o deskryptorze `fd`. Liczba przeczytanych bajtów równa jest rozmiarowi typu `unsigned long`, który wynosi 4 bajty dla linuksowych systemów 32-bitowych, a 8 bajtów dla systemów 64-bitowych. Funkcja zwraca liczbę przeczytanych bajtów lub `-1` w przypadku błędu:

```
unsigned long word;
ssize_t nr;

/* czytanie kilku bajtów z pliku 'fd' do bufora 'word' */
nr = read (fd, &word, sizeof (unsigned long));
if (nr == -1)
   /* błąd */
```

Przy zbyt prostej implementacji mogą pojawić się dwa problemy: funkcja może nie przeczytać wszystkich bajtów, których wymagana liczba została przekazana w parametrze `len`. Może ona również zwrócić takie błędy, których obsługa nie została uwzględniona w tej implementacji. Niestety, kod zbliżony do powyższego spotyka się bardzo często. Za chwilę zostanie pokazane, w jaki sposób można go ulepszyć.

Wartości zwracane

Funkcja `read()`podczas poprawnego wykonania może zwrócić dodatnią, niezerową liczbę, mniejszą od wartości parametru `len`. Istnieje kilka powodów takiego zachowania: w pliku było dostępnych mniej bajtów, niż zostało przekazanych w parametrze `len`, funkcja systemowa mogła zostać przerwana przez sygnał, mógł być uszkodzony potok (przy założeniu, że `fd` był jego deskryptorem) itd.

Podczas użycia funkcji `read()`, należy rozważyć również prawdopodobieństwo, że zwrócona wartość będzie równa zeru. Funkcja systemowa `read()` zwraca zero, by zasygnalizować znacznik końca pliku (EOF). W tym przypadku oczywiście nie będzie możliwe czytanie żadnych bajtów. EOF nie jest uważany za błąd (i dlatego też nie towarzyszy kodowi powrotu równemu `-1`); sygnalizuje jedynie, że pozycja w pliku przesunęła się poza jego koniec i nie ma już nic do odczytania. Jeśli jednak za pomocą funkcji należy przeczytać określoną liczbę bajtów (przekazaną w parametrze `len`), a nie ma do nich dostępu, funkcja zablokuje się (przełączy w tryb uśpienia) i będzie trwała w tym stanie, dopóki bajty nie staną się dostępne (przy założeniu, że deskryptor pliku nie został otwarty w trybie nieblokującym, opisanym w podrozdziale Odczyty nieblokujące). Należy zauważyć, że jest to inne zachowanie niż w przypadku zwracania EOF. Istnieje bowiem różnica między stanami „brak dostępnych danych", a „koniec danych". W przypadku EOF następuje osiągnięcie końca pliku. Gdy dochodzi do zablokowania, funkcja czytająca oczekuje na dalsze dane, na przykład podczas czytania z gniazda lub z pliku urządzenia.

Niektóre błędy można naprawić. Na przykład, gdy wywołanie funkcji `read()` będzie przed odczytaniem bajtów przerwane przez sygnał, zwrócona zostanie wartość `-1` (`0` mogłoby wprowadzić nieporozumienie z EOF), a zmienna `errno` ustawiona zostanie na `EINTR`. W tym przypadku wystarczy ponowić próbę czytania.

Po zakończeniu wywołania funkcji `read()` mogą powstać różne stany wyjściowe:

- Funkcja zwraca wartość równą wartości parametru `len`. Wszystkie bajty zostają odczytane i zapisane do bufora `buf`. Wyniki są takie, jak zaplanowano.
- Funkcja zwraca wartość mniejszą, niż wynosi parametr `len`, lecz większą od zera. Przeczytane bajty zostają zapisane do bufora `buf`. Taka sytuacja może się pojawić z kilku powodów: jakiś sygnał przerwał odczytywanie w trakcie jego trwania, wystąpił błąd odczytu, było dostępnych więcej niż zero bajtów, ale mniej niż wynosi wartość `len`, osiągnięto EOF, zanim wszystkie zakładane bajty zostały odczytane. Ponowna próba (z odpowiednio uaktualnionymi wartościami `buf` i `len`) spowoduje odczytanie pozostałych bajtów i zapisanie ich do dalszego fragmentu bufora lub wykaże przyczynę problemu.
- Funkcja zwraca `0`, co oznacza EOF. Nie ma żadnych bajtów do czytania.
- Funkcja zablokuje się, ponieważ aktualnie nie są dostępne żadne dane. Takie zjawisko nie zdarzy się w przypadku trybu nieblokującego.
- Funkcja zwróci `−1`, a zmienna `errno` zostanie ustawiona na `EINTR`. Jest to spowodowane pojawieniem się sygnału, zanim zostaną odczytane jakiekolwiek dane. W tym przypadku wywołanie funkcji może być ponowione.
- Funkcja zwróci `−1`, a zmienna `errno` zostanie ustawiona na `EAGAIN`. Takie zachowanie sygnalizuje wstrzymanie odczytu, ponieważ aktualnie nie ma dostępnych żadnych danych. Wywołanie funkcji może być ponowione po pewnym czasie. Problem ten pojawia się tylko w trybie nieblokującym.
- Funkcja zwróci `−1`, a zmienna `errno` zostanie ustawiona na wartość różną od `EAGAIN` lub `EINTR`. Sygnalizuje to poważniejszy błąd.

Czytanie wszystkich bajtów

Przedstawione powyżej warianty zakończenia funkcji `read()` uświadamiają, że wcześniej zaprezentowany przykładowy kod, który ją wywoływał, nie jest wystarczający w przypadku, gdy należy obsłużyć różne błędy i faktycznie odczytać cały ciąg bajtów o długości `len` (a przynajmniej odczytać wszystko, aż do pojawienia się EOF). Aby to wykonać, należy stworzyć pętlę wraz z paroma instrukcjami warunkowymi:

```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0)
{
   if (ret == -1)
   {
      if (errno == EINTR)
         continue;
      perror ("read");
      break;
   }

   len -= ret;
   buf += ret;
}
```

Ten fragment kodu obsługuje wszystkie pięć warunków. W pętli następuje odczytywanie bajtów do bufora `buf`, poczynając od aktualnej pozycji w pliku `fd`. Proces ten trwa do momentu, gdy liczba bajtów osiągnie wartość `len` lub gdy pojawi się EOF. Jeśli odczytano więcej niż zero,

lecz mniej niż `len` bajtów, następuje zmniejszenie wartości `len` i zwiększenie wskaźnika `buf` o liczbę już odczytanych bajtów, a następnie funkcja zostaje ponownie wywołana. Jeśli funkcja zwróci `-1`, a zmienna `errno` ustawiona zostanie na `EINTR`, czytanie jest ponawiane bez aktualizowania parametrów. Jeśli funkcja zwróci `-1`, a zmienna `errno` ustawiona zostanie na inną wartość, następuje wywołanie funkcji systemowej `perror()`, w celu wysłania komunikatu błędu do standardowego wyjścia błędów — program kończy pętlę.

Fragmentaryczne odczytywanie danych to praktyka nie tylko błędna, ale niestety powszechnie stosowana. Mnóstwo błędów w programach jest dziełem samych programistów, którzy niewłaściwie sprawdzali i obsługiwali krótkie żądania odczytów. Nie należy brać z nich przykładu!

Odczyty nieblokujące

Czasami programiści nie chcą wywoływać funkcji `read()` w taki sposób, aby blokowała się w czasie oczekiwania na dane. Zamiast tego chcieliby, aby funkcja natychmiast wracała po jej wywołaniu, sygnalizując, że dane nie są dostępne. Takie zachowanie zwane jest *nieblokującymi operacjami wejścia i wyjścia* (ang. *nonblocking I/O*); pozwala aplikacjom na wykonywanie operacji wejścia i wyjścia (również na wielu plikach) bez żadnego blokowania, które mogłoby spowodować pominięcie danych dostępnych w innym pliku.

Dlatego też należy sprawdzać dodatkową wartość zmiennej `errno`: `EAGAIN`. Jak już wcześniej napisano, jeśli deskryptor pliku zostanie otwarty w trybie nieblokującym (czyli w funkcji `open()` zostanie użyty znacznik `O_NONBLOCK`, jak opisano to w podrozdziale „Znaczniki w funkcji open()"), a jednocześnie nie istnieją dane do odczytania, funkcja `read()`, zamiast zostać zablokowana, zwróci natychmiast `-1`, a zmienna `errno` zostanie ustawiona na `EAGAIN`. Podczas wykonywania odczytów nieblokujących należy sprawdzać, czy pojawił się `EAGAIN`, gdyż może powstać poważny błąd i nastąpi utrata dużej ilości danych. Poniższy kod może służyć jako przykład odczytu danych w trybie nieblokującym:

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1)
{
   if (errno == EINTR)
      goto start; /* instrukcji goto nie powinno się używać ;) */
   if (errno == EAGAIN)
      /* po jakimś czasie należy ponowić próbę czytania */
   else
     /* błąd */
}
```

Obsługa błędu `EAGAIN` poprzez użycie instrukcji skoku bezwarunkowego `goto start` w powyższym przykładzie faktycznie nie ma sensu — zamiast niej można po prostu nie stosować nieblokujących operacji wejścia i wyjścia. Użycie instrukcji skoku bezwarunkowego zwiększa obciążenie procesora poprzez wykonywanie ciągłych pętli.

Inne wartości błędów

Inne wartości błędów dotyczą niewłaściwego zaprogramowania lub (dla EIO) problemów niskopoziomowych. Zmienna `errno` może zostać ustawiona na następujące wartości po błędnym wywołaniu funkcji `read()`:

`EBADF`
: Podany deskryptor pliku jest nieprawidłowy lub pliku nie otwarto do odczytu.

`EFAULT`
: Zmienna wskaźnikowa `buf` wskazuje na miejsce poza przestrzenią adresową wywołującego procesu.

`EINVAL`
: Deskryptor pliku jest odwzorowany na obiekt, który nie pozwala na czytanie.

`EIO`
: Wystąpił niskopoziomowy błąd wejścia i wyjścia.

`EISDIR`
: Deskryptor pliku odnosi się do katalogu.

Ograniczenia rozmiaru dla funkcji read()

Typy danych `size_t` oraz `ssize_t` są zdefiniowane przez POSIX. Typ `size_t` służy do przechowywania wartości określających rozmiar wyrażony w bajtach. Typ `ssize_t` jest po prostu typem `size_t` ze znakiem (ujemne wartości oznaczają błąd). W systemach 32-bitowych, pierwotnymi typami języka C są odpowiednio `unsigned int` oraz `int`. Tych dwóch typów danych często używa się wspólnie, dlatego też potencjalnie mniejszy zakres typu `ssize_t` wymusza wówczas ograniczenia zakresu dla typu `size_t`.

Maksymalną wartość dla typu `size_t` określa stała `SIZE_MAX`, natomiast dla typu `ssize_t` jest nią `SSIZE_MAX`. Jeśli wartość parametru `len` jest większa niż `SSIZE_MAX`, wówczas wynik wywołania funkcji `read()` jest nieokreślony. W większości systemów linuksowych, działających w 32-bitowej maszynie, `SSIZE_MAX` równa jest wartości `LONG_MAX`, która wynosi `0x7fffffff`. To stosunkowo dużo, jak na pojedynczą operację odczytu, ale mimo to należy o tym pamiętać. Aby usprawnić poprzednią procedurę odczytu, należałoby wykonać poniższy kod:

```
if (len > SSIZE_MAX)
   len = SSIZE_MAX;
```

Wywołanie funkcji `read()` z wartością parametru `len` równą zeru powoduje natychmiastowy powrót i zwrócenie zera.

Pisanie za pomocą funkcji write()

Najbardziej podstawową i powszechnie używaną funkcją systemową dla zapisu jest `write()`. Jest przeciwieństwem funkcji `read()` i została również zdefiniowana w POSIX.1:

```
#include <unistd.h>

ssize_t write (int fd, const void *buf, size_t count);
```

Wywołanie funkcji `write()` powoduje zapisanie bajtów, których liczba może wynosić `count` lub mniej. Dane pobierane są z obszaru pamięci, wskazywanego przez parametr `buf`, a zapisywane do pliku `fd`, począwszy od aktualnego położenia w nim. Pliki, które odwzorowują obiekty pozbawione możliwości wykonywania operacji szukania (np. urządzenia znakowe), zawsze zapisują „od początku".

W przypadku sukcesu, funkcja zwraca liczbę zapisanych bajtów, a położenie w pliku zostaje odpowiednio uaktualnione. W przypadku błędu, funkcja zwraca `-1`, a zmienna `errno` zostaje stosownie ustawiona. Wywołanie funkcji `write()` może zwrócić wartość `0`, lecz nie oznacza to żadnej ważnej informacji — po prostu informuje, że funkcja zapisała zero bajtów.

Tak jak w przypadku `read()`, podstawowy sposób użycia funkcji jest prosty:

```
const char *buf = "Mój statek jest mocny!";
ssize_t nr;

/* zapisz łańcuch znaków wskazywany przez 'buf' do pliku 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
   /* błąd */
```

Ponownie jak w przypadku `read()`, ten sposób użycia nie jest do końca poprawny. W procedurze wywołującej należy sprawdzić, czy nie nastąpił częściowy zapis:

```
unsigned long word = 1720;
size_t count;
ssize_t nr;

count = sizeof (word);
nr = write (fd, &word, count);
if (nr == -1)
   /* błąd, należy sprawdzić 'errno' */
else if (nr != count)
   /* prawdopodobny błąd, lecz zmienna 'errno' nie została ustawiona */
```

Zapisy częściowe

W porównaniu z funkcją `read()` i częściowym odczytem danych, funkcja systemowa `write()` rzadziej kończy się częściowym zapisem. Dla funkcji `write()` nie istnieje również warunek końca pliku (EOF). W przypadku plików zwykłych, funkcja `write()` gwarantuje wykonanie pełnego zapisu danych, chyba że wystąpi jakiś błąd.

Zgodnie z tym, dla plików zwykłych nie istnieje potrzeba wykonywania zapisów w pętli. Jednakże dla innych rodzajów plików, na przykład gniazd, pętla może być konieczna, by zagwarantować, że wszystkie wymagane bajty *rzeczywiście* zostaną zapisane. Inną korzyścią z użycia pętli jest to, że drugie wywołanie funkcji `write()` może zwrócić kod błędu, który poinformuje o przyczynie częściowego zapisu, powstałego podczas pierwszego wywołania funkcji (chociaż taka sytuacja występuje bardzo rzadko). Oto przykład:

```
ssize_t ret, nr;

while (len != 0 && (ret = write (fd, buf, len)) != 0)
{
   if (ret == -1)
   {
      if (errno == EINTR)
         continue;
      perror ("write");
```

```
            break;
        }
        len -= ret;
        buf += ret;
    }
```

Tryb dopisywania

Gdy plik `fd` zostaje otwarty w trybie dopisywania (poprzez użycie znacznika `O_APPEND`), zapis nie dokonuje się w miejscu aktualnego położenia w pliku. Wykonywany jest na końcu pliku.

Załóżmy na przykład, że dwa procesy zapisują dane do tego samego pliku. Gdyby procesy nie używały trybu dopisywania i pierwszy proces zapisywał dane na końcu pliku, a następnie drugi proces robił to samo, wówczas dla pierwszego procesu pozycja nie wskazywałaby już końca pliku. Zamiast tego pokazywałaby położenie równe końcowi pliku minus długość pakietu danych, które zostały właśnie zapisane przez drugi proces. Oznacza to, że w przypadku wielu procesów nie można nigdy dopisywać danych do tego samego pliku bez użycia jawnej synchronizacji, gdyż pojawia się zjawisko współzawodnictwa.

Tryb dopisywania chroni przed takimi problemami. Zapewnia, że pozycja w pliku jest zawsze ustawiona na końcu pliku, dlatego też każdy zapis dołączy dane poprawnie, nawet jeśli istnieje wiele procesów zapisujących. Można powiedzieć, że tryb dopisywania wykonuje aktualizację położenia w pliku przed każdym żądaniem zapisu. Położenie w pliku jest wówczas poprawnie uaktualnione, by wskazywać na koniec nowo dopisanych danych. Nie ma to znaczenia dla kolejnego wywołania funkcji `write()`, ponieważ automatycznie zaktualizuje ona położenie w pliku, lecz może mieć znaczenie, jeśli następnie (z jakichś powodów) wywołana zostanie funkcja `read()`.

Używanie trybu dopisywania ma sens w przypadku tylko niektórych zadań, np. aktualizacji plików z dziennikami zdarzeń.

Zapisy nieblokujące

Gdy plik `fd` otwarty jest w trybie nieblokującym (przy użyciu znacznika `O_NONBLOCK`), a wykonywany zapis w trybie zwykłym mógłby zablokować funkcję `write()`, następuje natychmiastowe wyjście z kodem powrotu `-1`, a zmienna `errno` ustawiona zostaje na `EAGAIN`. Żądanie zapisu powinno być ponowione w późniejszym czasie. Nie zdarza się to raczej w przypadku plików zwykłych.

Inne kody błędów

Zmienna `errno` może zawierać następujące znaczące kody błędów:

`EBADF`
: Podany deskryptor pliku jest niepoprawny lub nie został otwarty do zapisu.

`EFAULT`
: Zmienna wskaźnikowa `buf` wskazuje na miejsce poza przestrzenią adresową wywołującego procesu.

`EFBIG`
: Zapis mógłby spowodować utworzenie pliku, który miałby długość większą od maksymalnej dopuszczalnej wartości dla procesu lub od wewnętrznego ograniczenia implementacyjnego.

`EINVAL`
: Deskryptor pliku odwzorowany jest na obiekt, który nie pozwala na operację zapisu.

`EIO`
: Wystąpił niskopoziomowy błąd wejścia i wyjścia.

`ENOSPC`
: System plików, w którym znajduje się dany deskryptor pliku, nie posiada wystarczającego miejsca do przeprowadzenia operacji zapisu.

`EPIPE`
: Podany deskryptor pliku związany jest z potokiem lub gniazdem, którego wejście czytające jest zamknięte. Proces zapisujący otrzyma również sygnał `SIGPIPE`. Domyślna obsługa sygnału `SIGPIPE` powoduje przerwanie procesu czytającego. Dlatego też procesy otrzymują tę wartość `errno` wyłącznie wtedy, gdy jawnie proszone są o zignorowanie, zablokowanie lub obsługę sygnału `SIGPIPE`.

Ograniczenia rozmiaru dla funkcji write()

Jeśli parametr `count` jest większy niż `SSIZE_MAX`, wówczas wynik działania funkcji `write()` jest nieprzewidywalny.

Wywołanie funkcji `write()` z wartością parametru `count` równą zeru spowoduje, że funkcja zakończy się natychmiast i zwróci kod powrotu zero.

Sposób działania funkcji write()

W trakcie działania funkcji systemowej `write()`, jądro kopiuje dane z dostarczonego bufora do bufora jądra. Nie ma jednak żadnej gwarancji, że w momencie, gdy funkcja `write()` kończy swoje działanie, dane zostały już zapisane do docelowej lokalizacji. Faktycznie, powrót z funkcji `write()` odbywa się zbyt szybko, by całkowicie zakończyć zapis danych. Różnice wydajnościowe pomiędzy różnymi procesorami i dyskami twardymi mogą spowodować, że takie zachowanie będzie przyczyną problemów.

Gdy aplikacja z przestrzeni użytkownika wykonuje funkcję systemową `write()`, jądro Linuksa przeprowadza najpierw kilka testów, a następnie kopiuje dane do swojego bufora. Dopiero później, działając już w tle, jądro wyszukuje wszystkie „brudne" bufory, optymalnie je sortuje i zapisuje na dysk (proces ten nazywa się *opóźnionym zapisem* (ang. *writeback*). Pozwala to na wielokrotne wykonanie bardzo szybkich wywołań funkcji `write()`, które kończą swoje działanie prawie niemal natychmiast. Umożliwia to jądru również przeprowadzanie zapisu danych w momencie mniejszego obciążenia systemu, a także pozwala na wspólne przetwarzanie wielu zapisów.

Opóźnione zapisy nie zmieniają semantyki POSIX. Na przykład, jeśli nastąpi próba odczytu danych, które właśnie zostały zapisane i znajdują się dopiero w określonym buforze, a nie na dysku, wówczas zrealizowane zostanie to poprzez pobranie pakietu bajtów z tego bufora, a nie ze stałego miejsca na dysku. Zachowanie to w rzeczywistości poprawia wydajność, ponieważ

operacja czytania realizuje swoje zadanie za pomocą danych z bufora w pamięci, a nie z dysku. Żądania odczytu i zapisu przeplatają się wzajemnie, tak jak zaplanowano. Wyniki są takie, jakich oczekiwano — oczywiście o ile nie nastąpi załamanie systemu operacyjnego, zanim dane nie zostaną zapisane na dysk! W tym przypadku aplikacja będzie zakładać, że zapis zakończył się poprawnie, dane jednak nigdy nie pojawią się na dysku.

Innym problemem podczas operacji opóźnionego zapisu jest niezdolność do wymuszenia *kolejności zapisów*. Chociaż aplikacja może dbać o zachowanie porządku żądań zapisu poprzez zachowanie określonej kolejności wywołań funkcji zapisu, jądro samo uporządkuje te żądania, biorąc pod uwagę przede wszystkim uzyskanie lepszej wydajności działania. Jest to problemem tylko wtedy, gdy następuje załamanie systemu, gdyż w normalnych warunkach każdy z buforów w końcu zostaje zapisany na dysk i wszystko działa poprawnie. Nawet wtedy dla większości aplikacji nie jest istotna kolejność zapisów.

Ostatnim problemem, związanym z opóźnionymi zapisami, jest zgłaszanie pewnych błędów wejścia i wyjścia. Dowolny błąd wejścia i wyjścia, który wystąpi w trakcie zapisu danych z buforów na dysk (np. fizyczne uszkodzenie dysku), nie może być zgłoszony do procesu, który zażądał operacji zapisu. Faktycznie, bufory w ogóle nie są związane z procesami. Wiele procesów, które mogły istnieć tylko pomiędzy zapisem danych do pojedynczego bufora, a wyrzuceniem ich z niego na dysk, być może zmieniło już te dane. Poza tym, w jaki sposób można skontaktować się z procesem, którego operacja zapisu nie powiodła się już po zakończeniu wywołania żądania zapisu?

Jądro podchodzi poważnie do zagadnienia minimalizacji ryzyka wystąpienia opóźnionych zapisów. By upewnić się, że dane zostają zapisane na dysk w określonym czasie, jądro używa specjalnego parametru o nazwie *maksymalny wiek bufora* (ang. *maximum buffer age*) i zapisuje zdezaktualizowane bufory, zanim ich „czas życia" nie przekroczy tej wartości. Użytkownik ma możliwość modyfikacji tej wartości poprzez plik */proc/sys/vm/dirty_expire_centiseconds*. Wartość jest podawana w centysekundach (jedna setna sekundy).

Jest również możliwe, by wymusić opóźniony zapis dla danego bufora pliku lub nawet zsynchronizować wszystkie zapisy. Te zagadnienia omówione zostaną w następnym podrozdziale, zatytułowanym Zsynchronizowane operacje wejścia i wyjścia.

W podrozdziale Organizacja wewnętrzna jądra zostanie opisany w szczegółach podsystem jądra Linuksa, zapisujący dane z buforów na dysk (*system opóźnionego zapisu*).

Zsynchronizowane operacje wejścia i wyjścia

Choć synchronizacja operacji wejścia i wyjścia jest ważnym zagadnieniem, nie należy obawiać się problemów związanych z opóźnionymi zapisami. Buforowanie zapisów zapewnia *dużą* poprawę wydajności, dlatego też każdy system operacyjny, nawet w połowie zasługujący na nazwę: „nowoczesny", posiada zaimplementowaną obsługę opóźnionych zapisów poprzez bufory. Mimo tego są sytuacje, w których aplikacje powinny decydować o tym, kiedy dane mają zostać zapisane na dysk. W tych przypadkach Linux dostarcza kilku opcji, pozwalających zastąpić wydajność zsynchronizowanymi operacjami wejścia i wyjścia.

Funkcje fsync() i fdatasync()

Najprostszą metodą zapewnienia, że dane zostaną zapisane na dysk, jest użycie funkcji systemowej `fsync()`, zdefiniowanej w POSIX.1b:

```
#include <unistd.h>

int fsync (int fd);
```

Wywołanie funkcji `fsync()` zapewnia, że wszystkie dane, znajdujące się w zdezaktualizowanych buforach, powiązane z plikiem odwzorowanym na deskryptor `fd`, zostaną zapisane na dysk. Deskryptor pliku `fd` musi być otwarty w trybie zapisu. Funkcja zapisuje dane właściwe oraz metadane, takie jak informacje o czasie utworzenia pliku i inne atrybuty, znajdujące się w i-węźle. Funkcja ta nie zakończy się, dopóki nie otrzyma potwierdzenia od dysku twardego, że dane i metadane zostały na nim zapisane.

W przypadku gdy dysk twardy posiada bufor podręczny, `fsync()` nie ma możliwości sprawdzenia, czy dane zostały już fizycznie zapisane. Dysk twardy nie informuje, czy dane zostały już zapisane, czy też znajdują się jeszcze w jego buforze podręcznym. Na szczęście dane z bufora podręcznego na dysku twardym są szybko przenoszone na fizyczne medium magnetyczne.

Linux udostępnia również funkcję systemową `fdatasync()`:

```
#include <unistd.h>

int fdatasync (int fd);
```

Ta funkcja tylko opróżnia bufor danych; poza tym, jej działanie jest identyczne z działaniem funkcji `fsync()`. Nie gwarantuje ona więc, że nastąpi synchronizacja metadanych na dysku, przez co jej wywołanie może się szybciej zakończyć. Często jednak jej działanie jest wystarczające.

Obu funkcji można użyć w taki sam, bardzo prosty sposób:

```
int ret;

ret = fsync (fd);
if (ret == -1)
   /* błąd */
```

Żadna z tych funkcji nie gwarantuje, że dowolne, uaktualnione pozycje katalogu zawierające pliki, zostaną zsynchronizowane na dysku. Dlatego też, jeśli dowiązanie pliku zostało ostatnio uaktualnione, dane mogą zostać poprawnie zapisane na dysk. Nie dotyczy to związanej z tym plikiem pozycji katalogu i może spowodować, że plik stanie się niedostępny. By zapewnić, że dowolna aktualizacja pozycji w danym katalogu zostanie również zapisana na dysk, funkcja `fsync()` musi być jawnie wywołana również dla deskryptora pliku, będącego tymże katalogiem.

Wartości zwracane i kody błędów

W przypadku sukcesu, obie funkcje zwracają zero. W przypadku błędu, zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z trzech poniższych wartości:

`EBADF`
: Podany deskryptor pliku jest niepoprawnym deskryptorem, otwartym do zapisu.

`EINVAL`
: Podany deskryptor pliku jest odwzorowany na obiekt, który nie wspiera synchronizacji.

`EIO`
: Podczas synchronizacji wystąpił niskopoziomowy błąd wejścia i wyjścia. Ta wartość reprezentuje rzeczywisty błąd wejścia i wyjścia i jest często używana, aby przechwytywać tego typu błędy.

Wywołanie funkcji `fsync()` może się nie powieść w przypadku, gdy nie jest ona zaimplementowana w danym systemie plików, mimo że druga funkcja `fdatasync()` będzie zaimplementowana. Dla pełnego bezpieczeństwa aplikacje powinny użyć wywołania funkcji `fdatasync()`, gdy `fsync()` zwróci kod błędu `EINVAL`. Na przykład:

```
if (fsync (fd) == -1)
{
  /*
   * Lepiej użyć funkcji fsync(),
   * ale gdy ona nie działa,
   * należy wywołać fdatasync().
   */
  if (errno == EINVAL)
  {
     if (fdatasync (fd) == -1)
        perror ("fdatasync");
  }
  else
     perror ("fsync");
}
```

Ponieważ POSIX wymaga obecności funkcji `fsync()`, zaś funkcję `fdatasync()` sugeruje tylko jako opcję, dlatego też `fsync()` powinna być zawsze zaimplementowana dla plików zwykłych w każdym powszechnie używanym systemie plików Linuksa. Wyjątkiem są niestandardowe typy plików (być może takie, które nie posiadają metadanych dla synchronizacji) lub systemy plików, w których zaimplementowana może być tylko funkcja `fdatasync()`.

Funkcja sync()

Klasyczna funkcja systemowa `sync()` jest mniej optymalna, lecz wciąż szeroko używana. Wywołanie jej powoduje synchronizację *wszystkich* buforów na dysku:

```
#include <unistd.h>

void sync (void);
```

Funkcja nie posiada żadnych parametrów, ani nie zwraca żadnego kodu powrotu. Wykonuje się zawsze poprawnie, a jej zakończenie gwarantuje, że wszystkie bufory (zarówno dla danych, jak i metadanych) będą zapisane na dysku[4].

Standardy nie wymagają, aby funkcja `sync()` oczekiwała na zapis wszystkich buforów na dysk przed zakończeniem swojego działania; wymagane jest jedynie rozpoczęcie procesu zrzucania buforów na dysk. Z tego powodu często zalecane jest wielokrotne wykonanie synchronizacji, by upewnić się, że wszystkie dane znajdą się bezpiecznie na dysku. W Linuksie funkcja `sync()` *czeka*, dopóki wszystkie bufory nie będą zsynchronizowane na dysku. W tym przypadku jej jednokrotne wywołanie jest wystarczające.

[4] Pojawia się tu wspomniane już wcześniej zastrzeżenie: dysk twardy może „kłamać" i informować jądro, że bufory znajdują się już na dysku, mimo że w rzeczywistości będą ciągle w jego buforze podręcznym.

Jedynym faktycznym przypadkiem użycia funkcji systemowej `sync()` jest zastosowanie jej w programie użytkowym *sync(8)*. W aplikacjach należy używać funkcji `fsync()` oraz `fdata ↪sync()`, by wszystkie dane lub przynajmniej wymagane deskryptory mogły zostać zapisane na dysku. Należy zwrócić uwagę na to, że w przypadku obciążonych systemów wykonanie funkcji `sync()` może trwać nawet parę minut.

Znacznik O_SYNC

Użycie znacznika `O_SYNC` w wywołaniu funkcji `open()` wymusza *synchronizację* wszystkich operacji wejścia i wyjścia dla danego pliku:

```
int fd;

fd = open (file, O_WRONLY | O_SYNC);
if (fd == -1)
{
   perror ("open");
   return -1;
}
```

Żądania odczytu są zawsze synchronizowane. Gdyby tak nie było, wówczas poprawność odczytanych danych, znajdujących się w udostępnionym buforze, nie byłaby zagwarantowana. Jednakże, jak już wcześniej wspomniano, wywołania funkcji `write()` zwykle nie są synchronizowane. Nie istnieje więc związek między wykonaniem funkcji, a danymi zrzuconymi na dysk. Znacznik `O_SYNC` wymusza istnienie takiego związku, co powoduje, że funkcja `write()` działa w trybie zsynchronizowanych operacji wejścia i wyjścia.

Użycie znacznika `O_SYNC` sprawia, że przed każdym zakończeniem operacji `write()` wykonana zostanie funkcja `fsync()`. Istnieją oczywiście odpowiednie semantyki, choć jądro Linuksa implementuje znacznik `O_SYNC` trochę bardziej efektywnie.

Użycie tego znacznika dla operacji zapisu może spowodować pogorszenie parametrów *czasu użytkownika* oraz *czasu jądra* (czasu przebywania funkcji odpowiednio w przestrzeni użytkownika i przestrzeni jądra). Ponadto, w zależności od rozmiaru zapisywanego pliku, użycie `O_SYNC` może skutkować zwiększeniem całkowitego czasu wykonania funkcji o jeden lub nawet dwa rzędy wielkości, ponieważ koszty *całkowitego czasu oczekiwania dla operacji wejścia i wyjścia* (czyli czasu oczekiwania na zakończenie się tych operacji) ponosi proces. Są one znaczne, dlatego też zsynchronizowane operacje wejścia i wyjścia powinny być używane jedynie po wyczerpaniu się innych możliwych rozwiązań.

Zazwyczaj w aplikacjach wymagających gwarancji przesłania danych na dysk, używa się funkcji `fsync()` oraz `fdatasync()`. Generuje to mniejsze koszty niż w przypadku użycia znacznika `O_SYNC`, gdyż funkcje te są rzadziej wywoływane (tzn. tylko po wykonaniu pewnych krytycznych operacji).

Znaczniki O_DSYNC i O_RSYNC

W standardzie POSIX zdefiniowane są dwa inne znaczniki, również odnoszące się do zagadnienia synchronizacji operacji wejścia i wyjścia. Są to: `O_DSYNC` oraz `O_RSYNC`. W Linuksie te znaczniki są równoważne znacznikowi `O_SYNC`, co oznacza, że rezultaty ich działania w systemie są identyczne.

Znacznik O_DSYNC wskazuje, że po każdym zapisie powinna nastąpić wyłącznie synchronizacja zwykłych danych, a nie metadanych. Jest to równoważne ukrytemu wywołaniu funkcji fdatasync() po każdej operacji zapisu. Ponieważ znacznik O_SYNC zapewnia lepszą gwarancję wykonania operacji, dlatego też w przypadku, gdy jawnie nie stosuje się O_DSYNC, nie pojawia się żadne pogorszenie funkcjonalności; mogą ją pogorszyć jedynie wyższe wymagania, związane z użyciem znacznika O_SYNC.

Znacznik O_RSYNC wskazuje, że powinna wystąpić zarówno synchronizacja żądań odczytu, jak i zapisu. Znacznik ten musi zostać użyty razem z O_SYNC lub O_DSYNC. Jak już wcześniej wspomniano, odczyty są domyślnie zsynchronizowane — funkcja czytająca nie zakończy przecież swojego działania, dopóki nie będzie posiadała danych, które zostaną przekazane użytkownikowi. Użycie znacznika O_RSYNC powoduje, że również uboczne efekty procesu czytania zostaną zsynchronizowane. Oznacza to, że modyfikacje metadanych, spowodowane przeprowadzeniem operacji odczytu, muszą zostać zapisane na dysku, zanim funkcja zakończy swoje działanie. W praktyce zanim nastąpi powrót z funkcji read(), należy na dysku uaktualnić tylko wpis dla i-węzła, dotyczący ostatniego czasu dostępu do pliku. Znacznik O_RSYNC w Linuksie zdefiniowany jest tak samo jak O_SYNC, choć nie ma to sensu (te dwa znaczniki mają ze sobą tyle wspólnego, co O_SYNC z O_DSYNC). Obecnie w Linuksie nie ma możliwości, aby uzyskać zachowanie zdefiniowane przez znacznik O_RSYNC; w najlepszym wypadku programista może próbować wywołać funkcję fdatasync() po każdym wywołaniu funkcji read(). Taka procedura nie jest spotykana zbyt często.

Bezpośrednie operacje wejścia i wyjścia

Jądro Linuksa, podobnie jak jądra innych nowoczesnych systemów operacyjnych, implementuje złożoną warstwę oprogramowania dla obsługi buforowania i zarządzania operacjami wejścia i wyjścia pomiędzy urządzeniami a aplikacjami (opisane jest to w podrozdziale zatytułowanym Organizacja wewnętrzna jądra, znajdującym się przy końcu tego rozdziału). W aplikacji o dużej wydajności może być wymagane pominięcie tej złożonej warstwy i użycie własnego systemu zarządzania operacjami wejścia i wyjścia. Stworzenie takiego systemu nie jest jednak zwykle warte zachodu, gdyż w systemie operacyjnym istnieje wiele narzędzi, które pozwalają na osiągnięcie dużo lepszej wydajności niż ta osiągalna na zwykłym poziomie aplikacji. Mimo tego w systemach baz danych często używa się własnych metod buforowania i dąży do zminimalizowania obecności systemu operacyjnego.

Użycie znacznika O_DIRECT podczas wywołania funkcji open() wymusza na jądrze zminimalizowanie standardowego narzutu systemowego dla operacji wejścia i wyjścia. Gdy znacznik ten jest obecny, operacje wejścia i wyjścia będą zapisywać dane z buforów, znajdujących się w przestrzeni użytkownika, bezpośrednio na dysk, z pominięciem warstwy buforowania podręcznego. Wszystkie operacje wejścia i wyjścia będą synchroniczne; działanie funkcji zakończy się dopiero po ich zrealizowaniu.

Podczas przeprowadzania bezpośrednich operacji wejścia i wyjścia wymagana długość pakietu danych, wyrównanie bufora oraz położenie w pliku muszą być wartościami całkowitymi, będącymi wielokrotnością rozmiaru sektora dla obsługiwanego urządzenia — zwykle jest to 512 bajtów. Zanim pojawiła się wersja 2.6 jądra Linuksa, wymagania były bardziej surowe: dla wersji 2.4 wszystko musiało być dopasowane do rozmiaru logicznego bloku systemu plików (często równego 4 kB). W celu zachowania kompatybilności, aplikacje powinny dopasowywać się do większego (i niestety potencjalnie mniej wygodnego) rozmiaru bloku logicznego.

Zamykanie plików

Po zakończeniu pracy z deskryptorem pliku, program może go odłączyć od związanego z nim pliku poprzez użycie funkcji systemowej `close()`:

```
#include <unistd.h>

int close (int fd);
```

Podczas wywołania funkcji `close()` zamyka się deskryptor pliku `fd` oraz przerwane zostaje połączenie między procesem, a tym plikiem. Dany deskryptor pliku po tej operacji nie jest już prawidłowy i jądro może go ponownie wykorzystać w postaci wartości zwracanej podczas kolejnego wywołania funkcji `open()` lub `creat()`. Funkcja systemowa `close()` zwraca `0` w przypadku sukcesu. W przypadku błędu zwraca `-1` oraz odpowiednio ustawia zmienną `errno`. Użycie funkcji jest proste:

```
if (close (fd) == -1)
   perror ("close");
```

Należy zauważyć, że zamykanie pliku nie ma wpływu na to, kiedy dane zostaną przesłane na dysk. Jeśli aplikacja musi zapewnić, że plik będzie zapisany na dysk, zanim zostanie zamknięty, niezbędne okaże się użycie jednej z możliwych opcji synchronizacji, omówionej wcześniej w podrozdziale Zsynchronizowane operacje wejścia i wyjścia.

Zamykanie pliku powoduje jednak pewne efekty uboczne. Gdy zamknięty zostaje ostatni otwarty deskryptor danego pliku, struktura danych, reprezentująca plik wewnątrz jądra, zostaje usunięta. Podczas jej usuwania następuje również odłączenie od niej kopii i-węzła powiązanego z plikiem, znajdującej się w pamięci. Jeśli nic innego nie jest już podłączone do i-węzła, kopia ta może być również usunięta z pamięci (mimo to może ona pozostać dostępna, gdyż jądro buforuje same i-węzły z przyczyn wydajnościowych, natomiast kopie i-węzłów tego nie wymagają). Jeśli plik został odłączony od dysku, lecz był wcześniej cały czas otwarty, nie będzie on usunięty fizycznie, dopóki nie nastąpi jego zamknięcie, a i-węzeł nie zostanie usunięty z pamięci. Dlatego też wywołanie funkcji `close()` może również powodować, że odłączony plik zostanie fizycznie usunięty z dysku.

Kody błędów

Programiści powinni sprawdzać wartość zwracaną przez funkcję `close()`, co robią niestety rzadko. Może to spowodować pominięcie poważnego błędu, ponieważ nieprawidłowe sytuacje, związane z operacjami odroczonymi, mogą ujawniać się później, natomiast `close()` może informować o nich od razu.

Istnieje klika możliwych wartości zmiennej `errno`, ustawianych podczas niepowodzenia wykonania funkcji `close()`. Poza wartością `EBADF` (oznaczającą niepoprawny deskryptor pliku), najważniejszą wartością kodu błędu jest `EIO`, która informuje o niskopoziomowym błędzie wejścia i wyjścia, prawdopodobnie niezwiązanym z aktualną czynnością zamykania pliku. Bez względu na wartość błędu, deskryptor pliku (w przypadku gdy jest prawidłowy) zostaje zawsze zamknięty, a związane z nim struktury danych są usuwane z pamięci.

Funkcja systemowa `close()` nigdy nie zwróci wartości `EINTR`, choć zezwala na to POSIX. Zdecydowali tak projektanci jądra Linuksa, gdyż implementacja zgodna z POSIX-em nie byłaby elegancka.

Szukanie za pomocą funkcji lseek()

Zazwyczaj operacje wejścia i wyjścia w pliku wykonywane są w sposób liniowy, a domyślne aktualizacje położenia w pliku, spowodowane poprzez odpowiednie odczyty i zapisy, zaspokajają wszelkie potrzeby przeszukiwań. Niektóre aplikacje wymagają jednak nieliniowego poruszania się po pliku. Funkcja systemowa `lseek()` pozwala ustalić pozycję w pliku na określoną wartość. Poza zmianą pozycji w pliku nie wykonuje ona niczego innego; nie angażuje też żadnych operacji wejścia i wyjścia:

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek (int fd, off_t pos, int origin);
```

Zachowanie funkcji `lseek()` zależy od parametru `origin`, który może przyjmować następujące wartości:

`SEEK_CUR`
: Nowa pozycja w pliku o deskryptorze `fd` ustalona zostanie jako suma aktualnej pozycji oraz wartości `pos`. Wartość `pos` może być ujemna, dodatnia lub równa zeru. Jeśli `pos` będzie wynosić zero, pozycja w pliku nie zostanie zmieniona.

`SEEK_END`
: Nowa pozycja w pliku o deskryptorze `fd` ustalona zostanie jako suma długości pliku oraz wartości `pos`. Wartość `pos` może być ujemna, dodatnia lub równa zeru. Jeśli `pos` będzie wynosić zero, pozycja zostanie ustawiona na końcu pliku.

`SEEK_SET`
: Nowa pozycja w pliku o deskryptorze `fd` ustalona zostanie na wartość `pos`. Jeśli `pos` będzie wynosić zero, pozycja zostanie ustawiona na początku pliku.

W przypadku sukcesu, funkcja zwraca wartość nowego położenia w pliku. W przypadku błędu, zwraca `-1` oraz odpowiednio ustawia zmienną `errno`.

Na przykład, aby ustawić pozycję w pliku `fd` na wartość 1825, należy wykonać następujący kod:

```
off_t ret;

ret = lseek (fd, (off_t) 1825, SEEK_SET);
if (ret == (off_t) -1)
   /* błąd */
```

Poniższy kod ustawia pozycję w pliku `fd` na końcu pliku:

```
off_t ret;

ret = lseek (fd, 0, SEEK_END);
if (ret == (off_t) -1)
   /* błąd */
```

Ponieważ funkcja `lseek()` zwraca uaktualnioną wartość pozycji w pliku, może być użyta do zapytania o aktualną pozycję, poprzez wykonanie zerowego przesunięcia od pozycji `SEEK_CUR`:

```
int pos;

pos = lseek (fd, 0, SEEK_CUR);
if (pos == (off_t) -1)
   /* błąd */
else
   /* 'pos' zawiera wartość aktualnej pozycji pliku fd */
```

Jak dotąd, funkcji systemowej `lseek()` najczęściej używa się w następujących celach: przeszukanie pliku od początku, przeszukanie pliku od końca lub ustalenie aktualnej pozycji w pliku dla danego deskryptora pliku.

Szukanie poza końcem pliku

Możliwe jest nakazanie funkcji `lseek()` wykonania przesunięcia wskaźnika położenia poza koniec pliku. Na przykład, poniższy kod przeszukuje plik `fd`, 1688 bajtów, poza jego końcem:

```
int ret;

ret = lseek (fd, (off_t) 1688, SEEK_END);
if (ret == (off_t) -1)
  /* błąd */
```

Samo przeszukiwanie pliku poza jego końcem nic nie wnosi — próba odczytu od tego miejsca zwróci wartość EOF. Jeśli jednak dla tego położenia zostanie wykonana próba zapisu, stworzony zostanie nowy obszar danych, obejmujący zakres pomiędzy poprzednią a nową długością pliku. Obszar ten zostanie następnie wypełniony bajtami o wartości zero.

Uzupełnienie zerami zwane jest *luką* (ang. *hole*). W uniksopodobnych systemach plików luki nie zajmują fizycznie żadnego miejsca na dysku. Oznacza to, że całkowity rozmiar wszystkich plików w systemie plików może być większy, niż odpowiadający mu rozmiar fizycznie zapisanych danych na dysku. Pliki z lukami zwane są *plikami rzadkimi* (ang. *sparse files*). Mogą one zaoszczędzić dużo miejsca na dysku oraz poprawić ogólną wydajność działania systemu, gdyż praca z lukami nie wymaga generowania żadnych operacji wejścia i wyjścia.

Żądanie czytania z obszaru pliku, znajdującego się w luce, zwróci odpowiednią liczbę bajtów wypełnionych zerami.

Kody błędów

W przypadku błędu, funkcja `lseek()` zwraca wartość `-1` oraz ustawia zmienną `errno` na jedną z poniższych wartości:

`EBADF`
: Podany deskryptor pliku nie opisuje otwartego deskryptora pliku.

`EINVAL`
: Wartość podana w parametrze `origin` różni się od `SEEK_SET`, `SEEK_CUR`, `SEEK_END` *lub* obliczona pozycja w pliku jest wartością ujemną. Decyzja o reprezentowaniu obu typów błędów przez jeden kod `EINVAL` jest dość niefortunna. Pierwszy z tych typów przeważnie oznacza błąd programowania wykryty podczas kompilacji, drugi z kolei może oznaczać bardziej podstępny błąd logiki działania programu.

`EOVERFLOW`
: Wynikowa pozycja w pliku nie może być reprezentowana przez typ `off_t`. Problem ten może wystąpić jedynie w przypadku architektur 32-bitowych. W rzeczywistości pozycja w pliku *jest* aktualizowana; ten kod błędu informuje, że nie można jej zwrócić.

`ESPIPE`
: Podany deskryptor pliku związany jest z obiektem, który nie wspiera operacji szukania, takim jak potok, kolejka FIFO lub gniazdo.

Ograniczenia

Maksymalna wartość położenia w pliku ograniczona jest rozmiarem typu `off_t`. Dla większości architektur maszyn typ ten równy jest typowi `long` języka C, który w Linuksie posiada zawsze *rozmiar słowa* (dla danej maszyny jest to zwykle rozmiar roboczych rejestrów). Jednak wewnętrznie jądro przechowuje wartość przesunięcia w zmiennej typu `long long` dla języka C. Nie stanowi to problemu w maszynach 64-bitowych, ale oznacza to, że maszyny 32-bitowe podczas przeprowadzania przeszukiwań względnych mogą generować błędy `EOVERFLOW`.

Odczyty i zapisy pozycyjne

Aby nie używać funkcji `lseek()`, Linux dostarcza dwa warianty poprzednio omówionych funkcji systemowych `read()` i `write()` używających dodatkowego parametru, określającego położenie w pliku, od którego należy wykonywać operacje odczytu lub zapisu. Podczas powrotu, funkcje te *nie* aktualizują położenia w pliku.

Odmiana funkcji czytającej nazywana jest `pread()`:

```
#define _XOPEN_SOURCE 500

#include <unistd.h>

ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

Parametr `count` określa maksymalną liczbę bajtów do odczytu, umieszczonych w buforze `buf`, a czytanych z pliku `fd` od pozycji `pos`.

Odmiana funkcji zapisującej nazywana jest `pwrite()`:

```
#define _XOPEN_SOURCE 500

#include <unistd.h>

ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

Parametr `count` określa maksymalną liczbę bajtów do zapisu, które pobierane są z bufora `buf` i zapisywane do pliku `fd`, poczynając od pozycji `pos`.

Funkcje te są prawie identyczne w zachowaniu ze swoimi poprzednikami, których nazwy nie zaczynają się od litery „p”, za wyjątkiem całkowitego ignorowania ich aktualnego położenia w pliku; zamiast tego funkcje używają wartości przekazanej w parametrze `pos`. Również po wykonaniu nie aktualizują one położenia w pliku. Połączone użycie funkcji `read()` i `write()` mogłoby spowodować zniszczenie pracy wykonanej przez funkcje pozycyjne.

Obie funkcje pozycyjne mogą być używane jedynie z deskryptorami plików wspierającymi szukanie. Udostępniają one podobną semantykę jak poprzednie funkcje `read()` i `write()` użyte w połączeniu z funkcją `lseek()`. Mamy jednak trzy wyjątki. Po pierwsze, funkcje te są łatwiejsze w użyciu, szczególnie, gdy wykonuje się skomplikowane operacje, jak na przykład poruszanie się po pliku wstecz lub w porządku losowym. Po drugie, po użyciu tych funkcji nie następuje uaktualnienie położenia w pliku. Wreszcie, funkcje te zapobiegają powstawaniu potencjalnych sytuacji współzawodnictwa, które mogą występować podczas użycia `lseek()`. Ponieważ wątki dzielą miedzy sobą deskryptory plików, dlatego możliwe jest, aby inny wątek tego samego

programu uaktualnił pozycję w pliku zaraz po tym, jak pierwszy wątek wywołał funkcję `lseek()`, ale nie wykonał jeszcze funkcji `read()` i `write()`[5]. Takich sytuacji współzawodnictwa można uniknąć, stosując funkcje systemowe `pread()` i `pwrite()`.

Kody błędów

W przypadku sukcesu, obie funkcje zwracają liczbę odczytanych lub zapisanych bajtów. Wartość powrotu równa zeru w przypadku użycia funkcji `pread()` oznacza EOF; dla `pwrite()` oznacza ona, że funkcja nie zapisała żadnego bajtu. W przypadku błędu, obie funkcje zwracają `-1` oraz odpowiednio ustawiają zmienną `errno`. Funkcja `pread()` może ustawiać `errno` na każdą poprawną wartość, którą ustawiają funkcje `read()` i `lseek()`. Funkcja `pwrite()` może również ustawiać zmienną `errno` na każdą poprawną wartość, którą ustawiają funkcje `write()` i `lseek()`.

Obcinanie plików

Linux udostępnia dwie funkcje systemowe w celu zmiany długości pliku, które są zdefiniowane i wymagane (do pewnego stopnia) przez różne standardy POSIX. Oto one:

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate (int fd, off_t len);
```

oraz:

```
#include <unistd.h>
#include <sys/types.h>

int truncate (const char *path, off_t len);
```

Obie funkcje obcinają dany plik do długości podanej w parametrze `len`. Funkcja `ftruncate()` operuje na deskryptorze pliku, podanym w parametrze `fd`, który musi zostać otwarty do zapisu. Funkcja `truncate()` używa nazwy pliku podanej w parametrze `path`. Plik musi posiadać uprawnienia do zapisu. Obie funkcje zwracają wartość zero w przypadku sukcesu. W przypadku błędu, zwracają `-1` oraz odpowiednio ustawiają zmienną `errno`.

Najbardziej powszechnym przykładem użycia tych funkcji jest obcięcie pliku do rozmiaru mniejszego niż aktualnie istniejący. Po poprawnym wywołaniu funkcji długość pliku równa jest wartości parametru `len`. Dane, które wcześniej znajdowały się pomiędzy pozycją w pliku równą wartości `len` oraz poprzednią długością pliku, zostają zlikwidowane i nie są już dostępne poprzez funkcje odczytujące.

Funkcje te mogą być również użyte, by „obciąć" plik do rozmiaru dłuższego niż rozmiar pierwotny. Ich działanie jest wówczas podobne do przeszukiwania wraz z zapisem, co zostało już omówione wcześniej w podrozdziale Szukanie poza końcem pliku. Bajty dodane zostają wypełnione zerami.

Żadna z tych operacji nie modyfikuje aktualnej pozycji w pliku.

[5] Co spowoduje, że pierwszy wątek będzie czytał i pisał z innego miejsca w pliku, niż wcześniej zamierzał. Sytuacji takiej da się częściowo uniknąć, używając któregoś ze sposobów synchronizacji między wątkami — *przyp. tłum.*

Na przykład, załóżmy, że istnieje plik o nazwie *pirate.txt* i długości 80 bajtów, którego zawartość jest następująca:

```
Edward Teach był słynnym angielskim piratem.
Posiadał on przydomek Czarnobrody.
```

Później w tym samym katalogu należy uruchomić następujący program:

```
#include <unistd.h>
#include <stdio.h>

int main( )
{
   int ret;
   ret = truncate ("./pirate.txt", 45);
   if (ret == -1)
   {
      perror ("truncate");
      return -1;
   }
   return 0;
}
```

Program ten obetnie plik *pirate.txt* do długości 45 bajtów, co spowoduje, że jego aktualna zawartość będzie następująca:

```
Edward Teach był słynnym angielskim piratem.
```

Zwielokrotnione operacje wejścia i wyjścia

Wykonując wiele operacji wejścia i wyjścia, aplikacje często wymagają użycia większej liczby deskryptorów plików związanych z wejściem klawiatury (*stdin*), komunikacją międzyprocesową oraz zestawem plików. Nowoczesne aplikacje graficznego interfejsu użytkownika (GUI), sterowane zdarzeniami, mogą przetwarzać w głównej pętli programu[6] dosłownie setki oczekujących zdarzeń.

Bez pomocy wątków, oddzielnie obsługujących każdy deskryptor pliku, pojedynczy proces nie może prawidłowo blokować więcej niż jednego deskryptora pliku w tym samym czasie. Praca z wieloma deskryptorami plików wykonywana jest poprawnie, dopóki są one gotowe do zapisu lub odczytu. Ale gdy aplikacja będzie chciała użyć pewnego deskryptora pliku, który nie jest jeszcze gotowy do działania — np. została dla niego wywołana funkcja `read()`, która nie zwróciła jeszcze danych — proces zablokuje się i nie będzie już mógł obsługiwać innych deskryptorów plików. Blokada może potrwać parę sekund, co spowoduje, że aplikacja stanie się nieefektywna i będzie irytować użytkownika. Jednakże, gdy deskryptor pliku nie otrzyma żadnych danych, proces może zablokować się na stałe. Ponieważ operacje wejścia i wyjścia dla deskryptorów plików często są ze sobą powiązane (np. w mechanizmie potoków), jest całkiem możliwe, że dany deskryptor nie stanie się dostępny, dopóki nie zostanie obsłużony inny. Może to być całkiem poważnym problemem szczególnie dla aplikacji sieciowych, które jednocześnie używają wielu gniazd.

[6] Główna pętla powinna być znana tym, którzy zajmowali się tworzeniem aplikacji GUI — na przykład aplikacje GNOME używają głównej pętli udostępnianej przez podstawową bibliotekę GLib. Główna pętla pozwala na obserwowanie wielu zdarzeń i reagowanie na nie przy użyciu pojedynczego punktu blokowania.

Co będzie, gdy na wejściu *stdin* czekają dane do odczytania, a zablokowany zostanie deskryptor pliku dla komunikacji międzyprocesowej? Aplikacja nie będzie wiedziała, że wejście klawiatury czeka na obsługę, dopóki zablokowany deskryptor pliku IPC nie zwróci danych. A co będzie w przypadku, gdy zablokowana operacja nigdy się nie zakończy?

Wcześniej w rozdziale omówiono nieblokujące operacje wejścia i wyjścia, będące rozwiązaniem dla takich problemów. Przy użyciu nieblokujących operacji wejścia i wyjścia aplikacja może wysyłać żądania wejścia i wyjścia, które zwracają specjalny kod błędu, zamiast blokować się. Jednak z dwóch powodów rozwiązanie to nie jest efektywne. Po pierwsze, proces musi w dowolnej kolejności przez cały czas wykonywać operacje wejścia i wyjścia, czekając, aż jeden z jego otwartych deskryptorów plików będzie mógł na nie odpowiedzieć. Tak działa źle zaprojektowany program. Po drugie, system działałby o wiele sprawniej, gdyby program mógł przejść w stan uśpienia, pozwalając procesorowi wykonywać inne zadania, a następnie mógł być aktywowany jedynie wówczas, gdy jeden lub więcej deskryptorów plików będzie gotowych do wykonania operacji wejścia i wyjścia.

W tym celu należy użyć *zwielokrotnionych operacji wejścia i wyjścia* (ang. *multiplexed I/O*).

Zwielokrotnione operacje wejścia i wyjścia pozwalają aplikacji na równoczesne blokowanie się na wielu deskryptorach plików oraz na otrzymywanie powiadomienia, gdy tylko jeden z nich jest gotowy do odczytu lub zapisu bez blokowania. Zwielokrotnione operacje wejścia i wyjścia stają się więc centralnym miejscem w aplikacji, działając zgodnie z poniższym algorytmem:

1. Aplikacja wykorzystująca zwielokrotnione operacje wejścia i wyjścia „rozkazuje" systemowi, aby powiadamiał ją, gdy określone deskryptory plików będą gotowe do wykonania odczytów i zapisów.
2. Następnie aplikacja przechodzi w stan uśpienia do czasu, gdy jeden lub więcej deskryptorów plików będzie gotowych do wykonania operacji wejścia i wyjścia.
3. W pewnym momencie następuje aktywowanie aplikacji, która sprawdza, jakie deskryptory plików są gotowe.
4. Następuje obsługa w trybie nieblokującym wszystkich deskryptorów plików, gotowych do wykonania operacji wejścia i wyjścia.
5. Powrót do punktu 1.

Linux udostępnia trzy rozwiązania dotyczące zwielokrotnionych operacji wejścia i wyjścia. Są to interfejsy: *select*, *poll* oraz *epoll*. Dwa pierwsze zostaną omówione w tym rozdziale, a ostatni, będący zaawansowanym rozwiązaniem specyficznym dla Linuksa, przedstawiony w rozdziale 4.

Funkcja select()

Funkcja systemowa `select()` dostarcza mechanizmu pozwalającego na zaimplementowanie synchronicznych i zwielokrotnionych operacji wejścia i wyjścia:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
  struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
```

```
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Funkcja `select()` zostanie zablokowana do momentu, gdy dane deskryptory plików będą gotowe do wykonania operacji wejścia i wyjścia lub dopóki nie zostanie przekroczony opcjonalnie określony limit czasu.

Obserwowane deskryptory plików znajdują się w trzech tablicach, z których każda przyporządkowana jest innemu rodzajowi zdarzenia. Deskryptory plików, przekazane w parametrze `readfds`, obserwowane są w celu uzyskania informacji o możliwości czytania (oznacza to, że operacje odczytu będą mogły zostać wykonane w trybie nieblokującym). Deskryptory plików, przekazane w parametrze `writefds`, obserwowane są w celu uzyskania informacji o możliwości zapisu. W końcu deskryptory plików, przekazane w parametrze `exceptfds`, obserwowane są, by uzyskać informację, że nastąpiło wygenerowanie wyjątku lub dostępne są dane poza kolejką (dotyczy to tylko gniazd). Parametry, w których przekazywane są zestawy deskryptorów, mogą być równe `NULL`, co oznacza, że dla tych przypadków funkcja `select()` nie będzie oczekiwać na dane zdarzenie.

W przypadku poprawnego wykonania funkcji, każda z tablic zostanie zmodyfikowana w ten sposób, iż będzie zawierać tylko takie deskryptory plików, które są gotowe do przeprowadzenia operacji wejścia i wyjścia o określonym typie, wynikającym z przynależności do danej grupy. Na przykład, załóżmy istnienie dwóch deskryptorów plików o wartościach 7 i 9, umieszczonych w tablicy wskazywanej przez parametr `readfds`. Gdy funkcja się zakończy, a deskryptor 7 wciąż znajdować się będzie w tablicy, będzie to oznaczać, że jest on gotowy do przeprowadzenia nieblokującej operacji odczytu. Jeśli deskryptora 9 nie będzie w tablicy, oznacza to, że jest on prawdopodobnie niedostępny do odczytu w trybie nieblokującym (użyto określenia „prawdopodobnie", ponieważ jest możliwe, że dostęp do danych pojawi się po zakończeniu działania funkcji. W tym przypadku kolejne wywołanie funkcji `select()` zwróci deskryptor pliku gotowy do operacji odczytu[7]).

Parametr `n` równy jest wartości o jeden większej od największego identyfikatora deskryptora pliku dla wszystkich tablic. Zgodnie z tym, procedura wywołująca funkcję `select()` odpowiedzialna jest za sprawdzenie, który deskryptor pliku jest największy oraz za przekazanie zwiększonej o jeden wartości tego deskryptora w pierwszym parametrze danej funkcji.

Parametr `timeout` jest wskaźnikiem do struktury `timeval`, która została następująco zdefiniowana:

```
#include <sys/time.h>

struct timeval
{
   long tv_sec;  /* liczba sekund */
   long tv_usec; /* liczba mikrosekund */
};
```

Jeśli parametr ten nie jest równy wartości `NULL`, funkcja `select()` zakończy swoje działanie po liczbie sekund i mikrosekund równej odpowiednio wartościom pól `tv_sec` oraz `tv_usec`, nawet

[7] Dzieje się tak, ponieważ `select()` i `poll()` są operacjami aktywowanymi poziomem, a nie zboczem. Funkcja `epoll()`, która zostanie omówiona w rozdziale 4., może działać w obu trybach. Operacje aktywowane zboczem są prostsze, lecz mogą pomijać zdarzenia wejścia i wyjścia, jeśli się o to nie zadba.

gdy deskryptory plików już wcześniej będą gotowe do operacji wejścia i wyjścia. Po zakończeniu działania funkcji, zawartość struktury `timeval` jest nieokreślona dla różnych wersji systemu Unix, dlatego też musi zostać ona zawsze zainicjalizowana (włącznie z parametrami wskazującymi na tablice deskryptorów plików) przed każdym wywołaniem `select()`. W aktualnej wersji Linuksa struktura ta jest automatycznie modyfikowana i zawiera wartość określającą różnicę między limitem czasowym, a całkowitym czasem wykonania funkcji `select()`. Na przykład, jeśli limit czasowy został ustawiony na 5 sekund, a od momentu uruchomienia funkcji do otrzymania informacji o gotowości jakiegoś deskryptora pliku minęły 3 sekundy, wówczas po zakończeniu funkcji pole `timeval.tv_sec` będzie zawierało wartość 2.

Jeśli oba pola struktury limitu czasowego ustawione zostaną na wartość zero, funkcja zakończy się natychmiast, informując o zdarzeniach, które oczekiwały na obsługę w momencie jej wywołania, lecz nie czekając na wystąpienie kolejnych zdarzeń.

Tablice deskryptorów plików nie są modyfikowane bezpośrednio, lecz za pomocą pomocniczych makr. Pozwala to na dowolne zaimplementowanie tych tablic w różnych systemach uniksowych. Większość systemów implementuje je jednak jako zwykłe tablice bitowe. Makro `FD_ZERO` usuwa wszystkie deskryptory plików z danej tablicy. Powinno być ono użyte przed każdym wywołaniem funkcji `select()`:

```
fd_set writefds;

FD_ZERO(&writefds);
```

Makro `FD_SET` dodaje deskryptor pliku do danej tablicy, a FD_CLR usuwa deskryptor pliku z tablicy:

```
FD_SET(fd, &writefds); /* dodaj deskryptor pliku 'fd' do tablicy */
FD_CLR(fd, &writefds); /* usuń deskryptor pliku 'fd' z tablicy */
```

Dobrze zaprojektowany kod nie powinien nigdy używać makra `FD_CLR`. Jest ono rzadko (praktycznie wcale) używane.

Makro `FD_ISSET` sprawdza, czy dany deskryptor pliku znajduje się w określonej tablicy. Zwraca ono wartość niezerową, jeśli deskryptor jest zapisany w tablicy lub zero, jeśli go nie znaleziono. `FD_ISSET` używane jest po wywołaniu funkcji `select()`, aby sprawdzić, czy dany deskryptor pliku gotowy jest do działania:

```
if (FD_ISSET(fd, &readfds))
    /* deskryptor pliku 'fd' może być odczytany w trybie nieblokującym */
```

Ponieważ tablice deskryptorów plików są tworzone statycznie, istnieje ograniczenie dotyczące maksymalnej liczby deskryptorów plików umieszczanych w tych tablicach. Zdefiniowana jest również maksymalna wartość samego deskryptora. Ograniczenie to dla obu przypadków równe jest stałej `FD_SETSIZE` (w Linuksie ma ona wartość 1024). W dalszej części tego rozdziału omówione zostaną następstwa tych ograniczeń.

Wartości powrotu oraz kody błędów

W przypadku sukcesu, funkcja `select()` zwraca liczbę deskryptorów plików gotowych do przeprowadzenia operacji wejścia i wyjścia dla wszystkich trzech tablic łącznie. W przypadku błędu, funkcja zwraca `−1` oraz ustawia zmienną `errno` na jedną z poniższych wartości:

`EBADF`
: Jedna z tablic zawiera niepoprawny deskryptor pliku.

EINTR
: W czasie oczekiwania został przechwycony sygnał. Wywołanie funkcji może zostać powtórzone.

EINVAL
: Parametr `n` ma ujemną wartość lub podany limit czasowy jest nieprawidłowy.

ENOMEM
: Brak pamięci dla przeprowadzenia operacji.

Przykład użycia funkcji select()

Przedstawiony zostanie przykładowy program — trywialny, ale funkcjonalny. Ilustruje on użycie funkcji systemowej `select()`. Przykładowy kod blokuje się maksymalnie przez 5 sekund podczas oczekiwania na dane pojawiające się na wejściu *stdin*. Ponieważ obserwuje on jedynie pojedynczy deskryptor pliku, kod ten nie wykonuje faktycznie zwielokrotnionych operacji wejścia i wyjścia, lecz dzięki temu użycie funkcji systemowej `select()` przedstawione zostało w przejrzysty sposób:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define TIMEOUT 5 /* limit czasowy w sekundach */
#define BUF_LEN 1024 /* długość bufora odczytu */
int main (void)
{
   struct timeval tv;
   fd_set readfds;
   int ret;

   /* czekaj na pojawienie się danych na wejściu stdin */
   FD_ZERO(&readfds);
   FD_SET(STDIN_FILENO, &readfds);

   /* czekaj maksymalnie 5 sekund */
   tv.tv_sec = TIMEOUT;
   tv.tv_usec = 0;

   /* w porządku, teraz zablokuj się! */
   ret = select (STDIN_FILENO + 1, &readfds, NULL, NULL, &tv);
   if (ret == -1)
   {
      perror ("select");
      return 1;
   }
   else if (!ret)
   {
      printf ("Minęło %d sekund.\n", TIMEOUT);
      return 0;
   }
  /*
   * Czy używany deskryptor pliku gotowy jest do odczytu?
   * (Musi być, gdyż jest to jedyny deskryptor, jaki
   * został dostarczony do funkcji, która zwróciła wartość niezerową)
   */
   if (FD_ISSET(STDIN_FILENO, &readfds))
   {
      char buf[BUF_LEN+1];
      int len;
```

```
        /* zagwarantowane, że nie będzie blokować */
        len = read (STDIN_FILENO, buf, BUF_LEN);
        if (len == -1)
        {
           perror ("read");
           return 1;
        }

        if (len)
        {
           buf[len] = '\0';
           printf ("Przeczytano: %s\n", buf);
        }

        return 0;
     }

     fprintf (stderr, "Ten kod nie powinien się wykonać!\n");
     return 1;
  }
```

Przenośny sposób wstrzymania wykonania aplikacji za pomocą funkcji select()

Ponieważ funkcja `select()` była od dawna chętniej implementowana w różnych systemach Unix niż inne mechanizmy służące do precyzyjnego wstrzymania wykonania aplikacji, dlatego też stosuje się ją w celu uzyskania przenośnej metody przełączania programów w tryb uśpienia poprzez użycie parametru limitu czasowego różnego od `NULL`, a jednocześnie ustawienie parametrów dla wszystkich tablic na wartość `NULL`:

```
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = 500;

/* wstrzymaj wykonanie aplikacji przez 500 mikrosekund */
select (0, NULL, NULL, NULL, &tv);
```

Oczywiście Linux udostępnia interfejsy dla precyzyjnego wstrzymania wykonania aplikacji. Omówione zostaną one w rozdziale 10.

Funkcja pselect()

Funkcja systemowa `select()`, po raz pierwszy użyta w systemie 4.2BSD, jest popularna, lecz w standardzie POSIX zdefiniowano własne rozwiązanie — funkcję `pselect()`, która pojawiła się w POSIX 1003.1g-2000 oraz w POSIX 1003.1-2001:

```
#define _XOPEN_SOURCE 600
#include <sys/select.h>

int pselect (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
  const struct timespec *timeout, const sigset_t *sigmask);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Istnieją trzy różnice między funkcją `select()` a `pselect()`:

1. Typem parametru `timeout` dla funkcji `pselect()` jest struktura `timespec`, a nie `timeval`. Struktura `timespec` używa pól definiujących sekundy i nanosekundy, a nie sekundy i mikrosekundy, umożliwiając ustalenie limitu czasowego teoretycznie z bardzo dużą dokładnością.

W rzeczywistości jednak żadna funkcja nie pozwala na ustalenie czasu z dokładnością choćby do jednej mikrosekundy.

2. Wywołanie funkcji `pselect()` nie modyfikuje parametru `timeout`. Dlatego też nie musi być on ponownie inicjalizowany przed kolejnymi wywołaniami funkcji.
3. Funkcja systemowa `select()` nie posiada parametru `sigmask`. Gdy parametr ten zostanie ustawiony na wartość `NULL`, wówczas sposób działania funkcji `pselect()` w odniesieniu do sygnałów jest taki sam, jak system działania funkcji `select()`.

Struktura `timespec` zdefiniowana jest w następujący sposób:

```
#include <sys/time.h>

struct timespec
{
   long tv_sec;   /* liczba sekund */
   long tv_nsec; /* liczba nanosekund */
};
```

Podstawowym czynnikiem, zachęcającym do umieszczenia funkcji `pselect()` w zestawie narzędzi systemu Unix, był dodatkowy parametr `sigmask`, dzięki któremu możliwe było rozwiązanie problemu współzawodnictwa pomiędzy czekającymi deskryptorami plików, a sygnałami (sygnały zostaną dokładnie omówione w rozdziale 9.). Załóżmy, że procedura obsługi sygnału ustawia globalny znacznik (tak przeważnie jest), a proces sprawdza ten znacznik, zanim wywoła funkcję `select()`. Załóżmy też, że w danym przypadku sygnał wysłany zostaje już po sprawdzeniu znacznika, ale przed uruchomieniem funkcji. Aplikacja może przez to pozostać zablokowana na stałe i nie zareagować na ustawiony znacznik. Funkcja `pselect()` rozwiązuje ten problem poprzez umożliwienie aplikacji użycia tablicy sygnałów do zablokowania. Sygnały zablokowane nie są obsługiwane, dopóki się nie odblokują. Gdy tylko funkcja `pselect()` skończy swoje działanie, jądro odtworzy poprzednią maskę sygnałów. Dokładniejsze omówienie tego rozwiązania znajduje się w rozdziale 9.

Zanim pojawiła się wersja 2.6.16 jądra systemu, funkcja `pselect()` w Linuksie nie była funkcją systemową, lecz po prostu zwykłym opakowaniem funkcji `select()`, udostępnianym przez *glibc*. To opakowanie bardzo zmniejszało — lecz nie eliminowało całkowicie — ryzyko powstawania sytuacji współzawodnictwa. Od momentu wprowadzenia prawdziwej funkcji systemowej, problem współzawodnictwa przestał istnieć.

Pomimo usprawnień (stosunkowo niewielkich), wykonanych w funkcji `pselect()`, funkcja `select()`ciągle używana jest w większości aplikacji — zarówno z przyzwyczajenia, jak i z powodu większej przenośności.

Funkcja poll()

Funkcja systemowa `poll()` jest rozwiązaniem zwielokrotnionych operacji wejścia i wyjścia dla wersji Uniksa System V. Eliminuje ona kilka defektów funkcji `select()`, która jest jednak mimo tego ciągle używana (z przyzwyczajenia oraz większej przenośności):

```
#include <sys/poll.h>

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

W przeciwieństwie do funkcji `select()`, posiadającej nieeleganckie rozwiązanie w postaci trzech tablic bitowych przeznaczonych na deskryptory plików, funkcja `poll()` używa pojedynczej

tablicy struktur pollfd. Długość tablicy określana jest poprzez parametr nfds. Struktura pollfd zdefiniowana jest w następujący sposób:

```
#include <sys/poll.h>

struct pollfd
{
   int fd;          /* deskryptor pliku */
   short events;    /* zdarzenia oczekiwane */
   short revents;   /* zdarzenia, które wystąpiły */
};
```

Każda struktura pollfd określa pojedynczy deskryptor pliku, który powinien być obserwowany. Do funkcji poll() można przekazać więcej struktur, pozwalając jej na obserwowanie wielu deskryptorów plików. Pole events w każdej strukturze jest maską bitową zdarzeń, które oczekiwane są dla danego deskryptora pliku. Pole to zostaje ustawione przez użytkownika. Pole revents zawiera maskę bitową zdarzeń, które wystąpiły dla danego deskryptora pliku. To pole zostaje ustawione przez jądro podczas powrotu z funkcji. Wszystkie typy zdarzeń znajdujące się w polu events, mogą również pojawić się w polu revents. Istnieją następujące zdarzenia poprawne:

POLLIN
: Istnieją dane do odczytu.

POLLRDNORM
: Istnieją zwykłe dane do odczytu.

POLLRDBAND
: Istnieją priorytetowe dane do odczytu.

POLLPRI
: Istnieją pilne dane do odczytu.

POLLOUT
: Zapis danych nie będzie blokujący.

POLLWRNORM
: Zapis zwykłych danych nie będzie blokujący.

POLLWRBAND
: Zapis danych priorytetowych nie będzie blokujący.

POLLMSG
: Dostępna jest wiadomość SIGPOLL.

Następujące dodatkowe zdarzenia mogą zostać zwrócone w polu revents:

POLLER
: Wystąpił błąd w danym deskryptorze pliku.

POLLHUP
: Wystąpiło zawieszenie na danym deskryptorze pliku.

POLLNVAL
: Dany deskryptor pliku jest niepoprawny.

Te trzy ostatnie zdarzenia nie odnoszą się do pola events, gdyż są zwracane tylko w momencie ich wystąpienia. Podczas użycia funkcji poll(), w przeciwieństwie do select(), nie jest wymagane wykonanie jawnego zażądania zwracania informacji o zdarzeniach.

Kombinacja znaczników POLLIN | POLLPRI jest równoznaczna zdarzeniu odczytu dla funkcji select(), natomiast kombinacja POLLOUT | POLLWRBAND równoznaczna jest zdarzeniu zapisu dla tej funkcji. Znacznik POLLIN równy jest konstrukcji POLLRDNORM | POLLRDBAND, a POLLOUT równy jest POLLWRNORM.

Na przykład, aby obserwować deskryptor pliku w celu wykrycia możliwości zapisu i odczytu, należy użyć kombinacji zdarzeń POLLIN | POLLOUT. Po powrocie z funkcji należy sprawdzić, czy pole revents zawiera wymienione znaczniki. Ustawiony znacznik POLLIN informuje, że deskryptor pliku może zostać odczytany w trybie nieblokującym. Ustawiony znacznik POLLOUT oznacza, że deskryptor pliku może zostać użyty do zapisu bez blokowania. Znaczniki nie wykluczają się wzajemnie: oba mogą zostać ustawione, informując, że zarówno operacja zapisu, jak i odczytu zakończy się, zamiast blokować się na deskryptorze pliku.

Parametr timeout określa limit czasu oczekiwania (w milisekundach), po którym funkcja zakończy swoje działanie bez względu na to, czy pojawią się jakieś operacje wejścia i wyjścia do obsłużenia. Wartość ujemna oznacza nieskończony czas oczekiwania. Wartość zero informuje, że funkcja powinna zakończyć się natychmiast, zwracając listę tych deskryptorów plików, które w danym momencie posiadają oczekujące operacje wejścia i wyjścia. Funkcja nie czeka na pojawienie się kolejnych zdarzeń. Ten tryb pracy rzeczywiście odzwierciedla nazwę funkcji poll(), ponieważ wykonuje ona jednorazowe odpytanie i następnie od razu kończy swoje działanie.

Wartości powrotu oraz kody błędów

W przypadku sukcesu, funkcja poll() zwraca liczbę deskryptorów plików, których struktury zawierają niezerowe pola revents. Funkcja zwraca zero, jeśli upłynął limit czasowy, zanim pojawiło się jakieś zdarzenie. W przypadku błędu, funkcja zwraca −1 oraz ustawia zmienną errno na jedną z poniższych wartości:

EBADF
: Jedna lub więcej struktur zawiera niepoprawny deskryptor pliku.

EFAULT
: Parametr fds wskazuje na miejsce poza przestrzenią adresową wywołującego procesu.

EINTR
: W czasie oczekiwania na zdarzenia został odebrany sygnał. Wywołanie funkcji może zostać powtórzone.

EINVAL
: Parametr nfds przekroczył wartość RLIMIT_NOFILE.

ENOMEM
: Brak pamięci dla przeprowadzenia operacji.

Przykład użycia funkcji poll()

Pokazany zostanie przykładowy program używający funkcji poll() w celu jednoczesnego sprawdzania, czy operacje czytania z stdin oraz zapisywania do stdout będą blokować proces wywołujący:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/poll.h>
```

```
#define TIMEOUT 5 /* limit (w sekundach) czasu przepytywania */

int main (void)
{
   struct pollfd fds[2];
   int ret;

   /* obserwujemy wejście stdin */
   fds[0].fd = STDIN_FILENO;
   fds[0].events = POLLIN;

   /* obserwujemy wyjście stdout w celu stwierdzenia możliwości zapisu */
   /* (jest to prawie zawsze prawdą)*/
   fds[1].fd = STDOUT_FILENO;
   fds[1].events = POLLOUT;

   /* Wszystko ustawione, wywołajmy funkcję! */
   ret = poll (fds, 2, TIMEOUT * 1000);

  if (ret == -1)
    {
       perror ("poll");
       return 1;
    }

    if (!ret)
    {
       printf ("Minęło %d sekund.\n", TIMEOUT);
       return 0;
    }

    if (fds[0].revents & POLLIN)
       printf ("Wejście stdin jest gotowe do odczytu\n");

    if (fds[1].revents & POLLOUT)
       printf ("Wyjście stdout jest gotowe do zapisu\n");

    return 0;
}
```

Po uruchomieniu tego programu pojawi się oczekiwany wynik:

```
$ ./poll
Wyjście stdout jest gotowe do zapisu
```

Po ponownym uruchomieniu programu, podczas jednoczesnego skierowania zawartości jakiegoś pliku tekstowego na wejście `stdin`, wygenerowane zostaną oba zdarzenia:

```
$ ./poll < ode_to_my_parrot.txt
Wejście stdin jest gotowe do odczytu
Wyjście stdout jest gotowe do zapisu
```

Podczas użycia funkcji `poll()` w rzeczywistej aplikacji, nie jest wymagane ponowne tworzenie struktur `pollfd` przed każdym wywołaniem. Te same struktury mogą wielokrotnie być przekazywane do funkcji; jądro samo zadba o wyzerowanie pól `revents`.

Funkcja ppoll()

Linux udostępnia funkcję podobną do `poll()`, zwaną `ppoll()`. Używa ona dodatkowych parametrów, identycznych jak w przypadku funkcji `pselect()`. W przeciwieństwie do `pselect()`, funkcja `ppoll()` jest jednak specyficzna dla Linuksa:

```
#define _GNU_SOURCE
#include <sys/poll.h>

int ppoll (struct pollfd *fds, nfds_t nfds, const struct timespec *timeout,
  const sigset_t *sigmask);
```

Podobnie jak w przypadku `pselect()`, parametr `timeout` określa wartość limitu czasowego w sekundach i nanosekundach, a parametr `sigmask` dostarcza zestawu sygnałów, na które należy oczekiwać.

Porównanie funkcji poll() i select()

Chociaż obie te funkcje systemowe wykonują w zasadzie tę samą pracę, funkcja `poll()` jest z kilku powodów lepsza od funkcji `select()`:

- Funkcja `poll()` nie wymaga, aby użytkownik wyliczył i przekazał jako parametr funkcji wartość o jeden większą od największego numerycznie deskryptora pliku.
- Funkcja `poll()` jest bardziej efektywna w przypadku deskryptorów plików o dużych wartościach numerycznych. Na przykład podczas obserwacji pojedynczego deskryptora pliku o numerze 900 za pomocą funkcji `select()`, jądro musi sprawdzać wszystkie bity, aż do pozycji o numerze 900.
- Tablice deskryptorów plików dla funkcji `select()` posiadają statycznie ustalane rozmiary. Powoduje to, że w przypadku większego rozmiaru operacje na nich są nieefektywne, tym bardziej że nie wiadomo, czy te tablice są tablicami rzadkimi[8]. Z drugiej strony, użycie niedużych tablic ogranicza maksymalną wartość numeryczną dla deskryptora pliku, który może być obserwowany przy pomocy funkcji `select()`. Funkcja `poll()` pozwala na użycie tablic o dokładnie wymaganym rozmiarze. Gdy wymagana jest obserwacja jednego deskryptora, zostaje użyta tylko jedna struktura.
- W przypadku funkcji `select()` tablice deskryptorów plików są modyfikowane podczas jej powrotu, dlatego przed kolejnymi wywołaniami należy je znów inicjalizować. Funkcja systemowa `poll()` oddziela wejście (pole `events`) od wyjścia (pole `revents`), dzięki czemu tablica może zostać użyta bez potrzeby jej ponownej inicjalizacji.
- Parametr `timeout` dla funkcji `select()` posiada nieokreśloną wartość po jej zakończeniu. W kodzie przenośnym należy zadbać o to, aby ten parametr został ponownie zainicjalizowany. Problem ten nie istnieje w przypadku funkcji `poll()`.

Funkcja `select()` ma jednak kilka zalet:

- Jest bardziej przenośna, ponieważ funkcja `poll()` nie istnieje w niektórych wersjach systemu Unix.
- Zapewnia lepszą (w granicach mikrosekund) dokładność limitu czasowego. Teoretycznie, obie funkcje: `select()` i `poll()` zapewniają dokładność nanosekundową, lecz w rzeczywistości żadna z nich nie potrafi poprawnie ustalić limitu czasowego z dokładnością do mikrosekundy.

[8] Jeśli maska bitowa jest rzadka, wówczas każde słowo z tej maski może zostać sprawdzone, czy jego wartość jest równa zeru; tylko wtedy, gdy taka operacja zwróci fałsz, należy sprawdzić każdy z bitów tego słowa. Praca ta jest jednak bezużyteczna, gdy maska bitowa jest gęsto wypełniona wartościami różnymi od zera.

Oprócz tych dwóch funkcji istnieje jeszcze interfejs *epoll*, będący lepszym od nich, specyficznym dla Linuksa rozwiązaniem obsługi zwielokrotnionych operacji wejścia i wyjścia. Interfejs ten zostanie omówiony w rozdziale 4.

Organizacja wewnętrzna jądra

W tym podrozdziale omówiona zostanie wewnętrzna implementacja operacji wejścia i wyjścia dla Linuksa. Analizie poddane zostaną przede wszystkim trzy główne podsystemy jądra: *wirtualny system plików* (VFS), *bufor stron* oraz *opóźniony zapis stron*. Wspólne działanie tych trzech podsystemów umożliwia wykonywanie płynnych, efektywnych i optymalnych operacji wejścia i wyjścia.

W rozdziale 4. zostanie omówiony czwarty podsystem o nazwie *zarządca operacji wejścia i wyjścia* (ang. *I/O scheduler*).

Wirtualny system plików

Wirtualny system plików (VFS), zwany także czasami *przełącznikiem pliku wirtualnego* (ang. *virtual file switch*), jest abstrakcyjnym mechanizmem, dzięki któremu jądro Linuksa może używać różnych funkcji oraz modyfikować dane w określonym systemie plików, nie posiadając w ogóle wiedzy na temat jego typu i implementacji.

VFS realizuje tę abstrakcję poprzez udostępnienie *ogólnego modelu pliku*, który jest podstawą do tworzenia wszystkich systemów plików w Linuksie. Poprzez użycie wskaźników do funkcji oraz różnych, obiektowo zorientowanych procedur[9] ogólny model pliku udostępnia wzorzec, który musi być używany w jądrze Linuksa dla wszystkich systemów plików. Pozwala to wirtualnemu systemowi plików na stworzenie uogólnionych wywołań funkcji. Wzorzec udostępnia *punkty wywołania* (ang *hooks*) dla operacji odczytu, tworzenia dowiązań, synchronizacji itd. Następnie każdy system plików przeprowadza proces rejestrowania funkcji, aby poinformować jądro, że potrafi obsłużyć określone operacje.

Podejście to wymusza pewien stopień kompatybilności pomiędzy systemami plików. Na przykład, VFS operuje terminami takimi jak i-węzły, superbloki oraz pozycje w katalogu. System plików, który nie pochodzi od Uniksa i jest być może pozbawiony takich pojęć jak i-węzły, musi po prostu zostać dostosowany. I tak faktycznie się dzieje: Linux udostępnia bez żadnych problemów takie systemy plików jak FAT czy NTFS.

Korzyści z użycia VFS są ogromne. Pojedyncza funkcja może czytać z *każdego* systemu plików znajdującego się na *dowolnym* medium; pojedynczy program użytkowy może kopiować dane pomiędzy różnymi systemami plików. Wszystkie systemy plików udostępniają takie same pojęcia, te same interfejsy i funkcje systemowe. Wszystko po prostu działa — i to działa dobrze.

Gdy w aplikacji następuje wywołanie funkcji systemowej `read()`, powoduje to wykonanie interesujących operacji. Biblioteka języka C dostarcza definicji funkcji systemowej, która zamieniona zostaje na odpowiednią instrukcję pułapki w czasie kompilacji programu. Gdy tylko proces

[9] Tak w języku C.

z przestrzeni użytkownika zostaje przechwycony w jądrze, przechodząc przez procedurę obsługi funkcji systemowej `read()`, następuje ustalenie, jakie obiekty związane są z danym deskryptorem pliku. Potem jądro wywołuje odpowiednią funkcję odczytu, związaną z takim obiektem. Jest ona częścią kodu systemu plików. Funkcja ta realizuje swoje zadanie — na przykład, poprzez faktyczny odczyt fizycznych danych z systemu plików. Następnie zwraca te dane do funkcji `read()` z przestrzeni użytkownika, która z kolei powraca do procedury obsługi funkcji systemowej, kopiującej dane z powrotem do przestrzeni użytkownika. Tam funkcja `read()` kończy wreszcie swoją pracę, a proces dalej kontynuuje swoje działanie.

Skutki działania wirtualnych systemów plików są znaczące dla programistów systemowych. Nie muszą oni troszczyć się o typ systemu plików ani o rodzaj nośnika danych. Ogólne funkcje systemowe, takie jak `read()`, `write()` itd. mogą obsługiwać pliki dla każdego wspieranego systemu plików i rodzaju nośnika.

Bufor stron

Bufor stron jest obszarem pamięci, przeznaczonym do przechowywania danych z systemu plików, do których wystąpiły ostatnio żądania dostępu. Transfer danych do dysku jest bardzo wolny, szczególnie w porównaniu z prędkością działania nowoczesnych procesorów. Przechowywanie danych w buforze stron pozwala jądru na szybki dostęp do nich oraz na unikanie powtarzających się operacji dyskowych.

Implementacja bufora stron nadużywa definicji pojęcia *czasowej lokalizacji*, będącej jednym z rodzajów *właściwości lokalizacji*. Czasowa lokalizacja oznacza, że dla zasobu, do którego w pewnym momencie wystąpiło żądanie dostępu, istnieje wysokie prawdopodobieństwo ponownego dostępu w niedalekiej przyszłości. Opłacalne jest więc przeznaczenie pewnego obszaru pamięci w celu buforowania danych, gdyż zabezpiecza to w przyszłości przed powstawaniem obciążających system operacji dostępu do dysku.

Bufor stron jest pierwszym miejscem, które zostaje sprawdzone przez jądro podczas szukania danych z systemu plików. Jądro odwołuje się najpierw do podsystemu pamięci, by czytać dane z dysku tylko wtedy, gdy nie zostaną one odnalezione w buforze stron. Dlatego też gdy pewien fragment danych jest czytany po raz pierwszy, zostaje przesłany z dysku do bufora stron, a do aplikacji przekazywany jest już z tego bufora. Jeśli następuje ponowny odczyt tych danych, są one po prostu zwracane z bufora stron. Wszystkie operacje w sposób niewidoczny używają bufora stron, zapewniając tym samym, że dane będą zawsze aktualne i poprawne.

Rozmiar bufora stron dla Linuksa jest zmienny. Gdy pojawia się nowe dane, bufor stron powiększa się, konsumując coraz więcej dostępnej pamięci. Jeśli bufor stron w końcu zużyje całą dostępną pamięć i pojawi się nowe żądanie przydziału dodatkowego obszaru, bufor stron zostanie *obcięty* poprzez usunięcie najrzadziej używanych stron, aby zapewnić miejsce tym, do których dostęp jest częstszy. Obcinanie to realizowane jest w sposób płynny i automatyczny. Zmienny rozmiar bufora stron pozwala Linuksowi na użycie całej pamięci systemu i buforowanie tak dużej ilości danych, jak to tylko możliwe.

Często jednak zamiast wykonywania operacji obcinania bufora stron bardziej sensowne byłoby przerzucanie na dysk rzadko używanych obszarów danych, gdyż usuwanie często używanego fragmentu strony nie jest efektywne, ponieważ może on zostać ponownie wczytany do pamięci podczas kolejnego żądania odczytu. Przerzucanie danych pozwala jądru na przechowywanie ich na dysku i wykazywanie większej ilości pamięci, niż w rzeczywistości jest zainstalowane

w maszynie. Jądro Linuksa implementuje różne heurystyki w celu ustalenia równowagi między wielkością przerzucanych danych, a częstotliwością obcinania bufora stron (oraz innych zapasów w pamięci). Heurystyki te mogą decydować, czy należy przerzucać dane na dysk, zamiast wykonywać operację obcięcia bufora stron — szczególnie w przypadku, gdy przerzucane dane nie są używane.

Ustalanie równowagi pomiędzy przerzucaniem stron a obcinaniem bufora stron dostrajane jest przy pomocy pliku */proc/sys/vm/swappiness*. Ten wirtualny plik zawiera zapisany parametr, który może przyjmować wartości od zera do 100; domyślnie wynosi on 60. Wyższa wartość wymusza preferowanie takich działań, które pozwalają na utrzymywanie bufora stron w pamięci i częstsze przerzucanie plików na dysk. Niższa wartość ustala preferencje dla obcinania bufora stron i zmniejszenia częstotliwości przerzucania plików na dysk.

Innym typem właściwości lokalizacji jest *lokalizacja sekwencyjna*, która polega na tym, że dostęp do danych występuje w formie sekwencyjnej. By wykorzystać tę zasadę, jądro implementuje również bufor stron dla *odczytu z wyprzedzeniem* (ang. *readahead*). Odczyt z wyprzedzeniem polega na czytaniu dodatkowych danych z dysku do bufora stron po każdym żądaniu odczytu — w rzeczywistości odczytuje się „zbyt wiele" danych. Gdy jądro czyta fragment danych z dysku, wykonuje zarazem odczyt kolejnych, dodatkowych porcji. Jednorazowy odczyt dużych, sekwencyjnie ułożonych fragmentów danych jest efektywny, ponieważ zwykle dysk nie musi wykonywać operacji szukania. Dodatkowo jądro może realizować żądania odczytu z wyprzedzeniem, podczas gdy proces zajęty jest przetwarzaniem pierwszego fragmentu odczytanych danych. Jeśli (co często się zdarza) proces wysyła nowe żądanie odczytu kolejnej porcji danych, jądro może je pobrać z bufora stron dla odczytu z wyprzedzeniem, zamiast wykonać operację wejścia i wyjścia na dysku.

Bufor dla mechanizmu odczytu z wyprzedzeniem, podobnie jak bufor stron, zarządzany jest w sposób dynamiczny. Jeśli jądro zauważa, że dany proces korzysta wciąż z danych przychodzących z tego bufora, następuje powiększenie okna odczytu, dzięki czemu pojawia się ich jeszcze więcej. Okno odczytu może mieć różne rozmiary, poczynając od 16 kB, a kończąc na 128 kB. I odwrotnie, jeśli jądro zauważa, że bufor dla mechanizmu odczytu z wyprzedzeniem nie jest wykorzystywany (gdy aplikacja nie czyta sekwencyjnie z pliku, lecz używa funkcji szukania), może ten mechanizm całkowicie zablokować.

Bufor stron powinien być niewidoczny. Programiści systemowi zwykle nie mogą usprawniać swojego kodu w taki sposób, by wykorzystywać obecność bufora stron, w przeciwieństwie do sytuacji, gdyby sami implementowali taki mechanizm w przestrzeni użytkownika. Stworzenie wydajnego kodu jest zwykle jedynym wymaganym warunkiem, aby najbardziej optymalnie używać bufora stron. Z drugiej strony, wykorzystanie odczytu z wyprzedzeniem jest możliwe. Sekwencyjne operacje plikowe wejścia i wyjścia są zawsze przedkładane nad operacjami z dostępem swobodnym, choć nie zawsze możliwe do zrealizowania.

Opóźniony zapis stron

Jak napisano już w podrozdziale Sposób działania funkcji write(), jądro opóźnia operacje zapisów na dysk poprzez użycie buforowania. Gdy proces wysyła żądanie zapisu, następuje skopiowanie danych do bufora, który staje się *buforem brudnym*, co oznacza, iż kopia danych w pamięci jest nowsza od kopii znajdującej się na dysku. Następnie funkcja zapisująca po prostu kończy swoje

działanie. Jeśli wystąpiło inne żądanie zapisu dotyczące tego samego fragmentu pliku, bufor zostaje zapełniony nowymi danymi. Żądania zapisu, wykonywane w różnych miejscach tego samego pliku, powodują powstawanie nowych buforów.

Ostatecznie bufory brudne muszą zostać zapisane, synchronizując dane z pamięci z danymi na dysku. Proces ten znany jest pod nazwą opóźnionego zapisu stron (*writeback*). Pojawia się on w przypadku spełnienia jednego z dwóch następujących warunków:

- Gdy wielkość dostępnej pamięci staje się mniejsza od pewnego konfigurowalnego poziomu granicznego, bufory brudne zapisane zostają na dysk, a nowo powstałe bufory czyste mogą zostać usunięte, zwalniając pamięć.
- Gdy czas życia bufora brudnego przekracza pewien konfigurowalny poziom graniczny, zostaje on zapisany na dysk. Chroni to dane przed pozostaniem na stałe w stanie brudnym.

Opóźnione zapisy stron obsługiwane są przez pakiet wątków jądra, znany pod nazwą *wątków pdflush* (nazwa pochodzi prawdopodobnie od angielskich słów *page dirty flush*, lecz kto wie, jak jest naprawdę). Gdy jeden z dwóch poprzednio opisanych warunków jest spełniony, następuje aktywowanie wątków pdflush i rozpoczęcie zrzucania buforów brudnych na dysk, do momentu aż żaden z tych warunków nie będzie już prawdziwy.

Może istnieć wiele wątków pdflush, które będą jednocześnie uruchamiać procedury opóźnionego zapisu. Jest to zrealizowane dzięki eksploatowaniu korzyści wynikających z równoległego przetwarzania oraz z zaimplementowania funkcji *unikania przeciążenia* (ang. *congestion avoidance*). Funkcja unikania przeciążenia zapewnia, że żądania zapisów nie będą oczekiwać w kolejce do tego samego urządzenia blokowego. Jeśli dla różnych urządzeń blokowych istnieją bufory brudne, uaktywnione zostaną odpowiednie wątki pdflush, aby w pełni wykorzystać każde z tych urządzeń. Likwiduje to niedoskonałość, istniejącą w starszych wersjach jądra: poprzednik wątków pdflush (*bdflush*, będący pojedynczym wątkiem) mógł poświęcić cały swój czas na oczekiwanie na pojedyncze urządzenie blokowe, podczas gdy inne urządzenia blokowe były nieobciążone. Aktualne wersje jądra Linuksa mogą jednocześnie „obciążać" dużą liczbę dysków.

Bufory reprezentowane są w jądrze poprzez strukturę o nazwie `buffer_head`. Ta struktura danych służy do obserwowania różnych metadanych związanych z buforem, jak na przykład tego, czy bufor jest czysty, czy brudny. Zawiera również wskaźnik do obszaru z rzeczywistymi danymi. Dane te umieszczone są w buforze stron. Dzięki temu podsystem buforowania dla opóźnionych zapisów oraz bufor stron są ujednolicone.

We wczesnych wersjach jądra Linuksa (przed wersją 2.4) podsystem buforowania dla opóźnionych zapisów był oddzielony od podsystemu buforowania stron, dlatego też istniał zarówno bufor stron, jak i bufor opóźnionych zapisów. Powodowało to, że dane mogły jednocześnie istnieć zarówno w buforze opóźnionych zapisów (jako dane bufora brudnego), jak i w buforze stron (jako dane zbuforowane). Synchronizacja tych dwóch oddzielnych systemów buforowania wymagała pewnego wysiłku. Ujednolicony bufor stron, wprowadzony w wersji 2.4 jądra Linuksa, był mile widzianym udoskonaleniem systemu.

Opóźnione zapisy oraz podsystem buforowania w Linuksie umożliwiają wykonywanie szybkich operacji zapisu, kosztem zwiększonego ryzyka utraty danych podczas zaniku zasilania. By zabezpieczyć się przed tym problemem, należy używać zsynchronizowanych operacji wejścia i wyjścia (omówionych wcześniej w tym rozdziale) w przypadku tworzenia aplikacji krytycznych.

Zakończenie

W rozdziale tym poddano analizie podstawy programowania systemowego w Linuksie: plikowe operacje wejścia i wyjścia. W Linuksie, będącym systemem operacyjnym, który dąży do reprezentowania wszystkiego, co tylko może wystąpić w postaci plików, bardzo ważna jest wiedza na temat ich otwierania, zapisywania, odczytywania oraz zamykania. Wszystkie te operacje dotyczą klasycznego Uniksa i są zdefiniowane w różnych standardach.

W następnym rozdziale omówione zostaną buforowane operacje wejścia i wyjścia oraz typowe interfejsy wejścia i wyjścia dla standardowej biblioteki języka C. Standardowa biblioteka języka C nie jest wyłącznie udogodnieniem; buforowane operacje wejścia i wyjścia w przestrzeni użytkownika powodują poważną poprawę wydajności.

ROZDZIAŁ 3.

Buforowane operacje wejścia i wyjścia

Jak napisano w rozdziale 1., wspólnym elementem dla wszystkich operacji wejścia i wyjścia jest *blok*, będący abstrakcją systemu plików. Podczas każdej operacji dyskowej używane jest pojęcie bloku. Zgodnie z tym, sprawność operacji wejścia i wyjścia jest najwyższa, gdy czytane i zapisywane pakiety danych posiadają granice adresów oraz wielkości równe wielokrotności rozmiaru bloku.

Spadek sprawności wynika ze zwiększonej liczby wywołań funkcji systemowych, które przykładowo próbują czytać po jednym bajcie z obszaru danych o rozmiarze 1 kB, zamiast przeczytać te 1024 bajtów od razu. Nawet ciąg operacji, przeprowadzanych przy użyciu pakietów o rozmiarach większych niż rozmiar bloku, może nie być zoptymalizowany, jeśli nie są one równe całkowitej wielokrotności wielkości bloku. Na przykład, jeśli rozmiar bloku wynosi 1 kB, wówczas wykonywanie operacji przy użyciu pakietów o wielkości 1130 bajtów może w dalszym ciągu trwać dłużej, niż w przypadku pakietów o rozmiarze równym 1024 bajtów.

Operacje wejścia i wyjścia, buforowane w przestrzeni użytkownika

Programy, które muszą wykonywać dużą liczbę prostych operacji wejścia i wyjścia dla plików zwykłych, często implementują *operacje wejścia i wyjścia, buforowane w przestrzeni użytkownika*. Oznacza to, że buforowanie wykonywane jest w przestrzeni użytkownika — jawnie przez aplikację lub automatycznie poprzez funkcje biblioteczne, nie dopuszczając jądra do tej operacji. Jak zostało to już omówione w rozdziale 2., jądro z powodów wydajnościowych buforuje wewnętrznie dane poprzez opóźnianie zapisu, łącząc w jedną całość kilka sąsiadujących ze sobą operacji wejścia i wyjścia, a także poprzez czytanie z wyprzedzeniem. Buforowanie w przestrzeni użytkownika, realizowane za pomocą innych metod, ma również na celu zwiększenie wydajności.

Oto przykład użycia programu o nazwie *dd*, pochodzącego z przestrzeni użytkownika:

```
dd bs=1 count=2097152 if=/dev/zero of=pirate
```

Dzięki użyciu parametru `bs` równemu `1`, wykonanie tej instrukcji spowoduje skopiowanie 2 megabajtów z urządzenia */dev/zero* (będącego urządzeniem wirtualnym, udostępniającym nieskończony ciąg zer) do pliku *pirate*, przy użyciu 2 097 152 jednobajtowych pakietów danych. Oznacza to, że do skopiowania danych wymagane będzie wykonanie około dwóch milionów operacji czytania i pisania — na jedną operację przypadać będzie jeden bajt.

Rozważmy teraz wykonanie tej samej operacji kopiowania 2 megabajtów danych, lecz przy użyciu bloków o rozmiarze 1024 bajtów:

```
dd bs=1024 count=2048 if=/dev/zero of=pirate
```

Operacja ta również spowoduje skopiowanie 2 megabajtów danych do tego samego pliku, lecz tym razem przy użyciu 1024 razy mniejszej liczby wywołań funkcji odczytu i zapisu. Poprawa wydajności jest olbrzymia, co potwierdzają wyniki przedstawione w tabeli 3.1. Zaprezentowano tam czas wykonania (mierzony za pomocą trzech różnych miar) trzech instrukcji *dd*, które różniły się między sobą tylko rozmiarem bloku. Czas właściwy oznacza całkowity czas wykonania instrukcji, czas użytkownika to czas wykonania kodu programu w przestrzeni użytkownika, natomiast czas systemu jest czasem wykonania funkcji systemowych w przestrzeni jądra, wywołanych przez proces programu *dd*.

Tabela 3.1. Wpływ rozmiaru bloku na wydajność

Rozmiar bloku	Czas właściwy	Czas użytkownika	Czas systemu
1 bajt	18,707 sekund	1,118 sekund	17,549 sekund
1024 bajtów	0,025 sekund	0,002 sekund	0,023 sekund
1130 bajtów	0,035 sekund	0,002 sekund	0,027 sekund

Użycie pakietów o rozmiarze 1024 bajtów pozwala na uzyskanie *olbrzymiej* poprawy wydajności w porównaniu z wykorzystaniem pakietów o rozmiarze pojedynczego bajta. Jednak z przedstawionych wyników można również wywnioskować, że użycie większego rozmiaru bloku (co wiąże się z wykonaniem nawet mniejszej liczby wywołań funkcji) może powodować spadek wydajności, jeśli w operacjach nie zostaną zastosowane pakiety danych o rozmiarach równych wielokrotności rozmiaru bloku. Pomimo mniejszej liczby wywołań funkcji, czas wykonania instrukcji przy użyciu pakietów o długości 1130 bajtów jest dłuższy od czasu wykonania takiej samej instrukcji, lecz używającej pakietów o długości 1024 bajtów. Wynika to z tego, że pakiety o rozmiarze 1130 bajtów nie są wyrównane do wielkości bloku i tym samym używanie ich jest mniej efektywne niż pakietów1024-bajtowych.

Wykorzystanie tych właściwości wymaga wiedzy na temat przypuszczalnego rozmiaru bloku fizycznego. Na podstawie wyników, przedstawionych w tabeli, można sądzić, że rozmiar bloku równy jest (w bajtach) najprawdopodobniej jednej z wymienionych wartości: 1024, wielokrotność liczby 1024, podzielnik liczby 1024. W przypadku urządzenia */dev/zero* rozmiar bloku wynosi w rzeczywistości 4096 bajtów.

Rozmiar bloku

W praktyce stosuje się następujące rozmiary bloków: 512, 1024, 2048, 4096 (wszystkie wyrażone w bajtach).

Jak zostało to już pokazane w tabeli 3.1., duża poprawa wydajności realizowana jest w prosty sposób poprzez wykonywanie operacji używających pakietów, których wielkość jest całkowitą

wielokrotnością lub podzielnikiem rozmiaru bloku. Dzieje się tak dlatego, ponieważ zarówno jądro, jak i sprzęt używają bloków danych podczas operacji wejścia i wyjścia. Z tego powodu wielkość pakietu, wyrównana do rozmiaru bloku, gwarantuje dopasowanie operacji wejścia i wyjścia i zabezpiecza przed dodatkową pracą, która musiałaby być wykonana przez jądro.

Określenie rozmiaru bloku dla danego urządzenia jest proste dzięki użyciu funkcji systemowej `stat()` (opisanej w rozdziale 7.) lub komendy *stat(1)*. Okazuje się jednak, że zwykle nie ma potrzeby, aby znać faktyczny rozmiar bloku.

Podstawową zasadą podczas wyboru rozmiaru pakietu dla operacji wejścia i wyjścia jest to, aby nie decydować się na niestandardową wartość, jak na przykład 1130. Żaden blok w historii Uniksa nie posiadał nigdy rozmiaru 1130 bajtów, tak więc wybór tego rozmiaru dla aplikacji z przestrzeni użytkownika spowoduje wykonanie takich operacji wejścia i wyjścia, dla których wielkość pakietu nie będzie wyrównana do rozmiaru bloku już po pierwszym wywołaniu funkcji zapisu lub odczytu. Użycie dowolnej wielokrotności lub podzielnika rozmiaru bloku zabezpiecza jednak przed takimi problemami. Dopóki wybrany rozmiar będzie wyrównany do wielkości bloku, wydajność pozostanie na dobrym poziomie. Wyższe wielokrotności spowodują, że konieczne będzie użycie mniejszej liczby wywołań funkcji systemowych.

Dlatego też najprostszym wyborem jest decyzja o użyciu dużego bufora w celu przeprowadzania operacji wejścia i wyjścia. Rozmiar bufora powinien być wielokrotnością typowego rozmiaru bloku. Doskonale pasuje do tego celu rozmiar o wielkości 4096, jak również 8192 bajtów.

Problem polega na tym, że w programach rzadko używane jest pojęcie bloku. W aplikacjach używa się pól, wierszy, pojedynczych znaków, a nie takich abstrakcji jak bloki. Jak już wcześniej napisano, by zapobiec takiej sytuacji, wprowadza się pojęcie operacji wejściowych i wyjściowych, buforowanych w przestrzeni użytkownika. Podczas zapisywania danych następuje najpierw ich zapamiętanie w buforze wewnątrz przestrzeni adresowej programu. Gdy dane w buforze osiągną pewien określony rozmiar — *rozmiar bufora* — zostaną w całości zapisane na dysk w trakcie pojedynczej operacji zapisu. W podobny sposób do bufora odczytu wczytywane są pakiety danych, wyrównane do rozmiaru bloku. Gdy następnie aplikacja chce czytać dane z tego bufora przy użyciu pakietów o niestandardowej wielkości, po kolei są one z niego pobierane, aż do momentu, gdy nastąpi jego opróżnienie. Wówczas do bufora wczytany zostaje kolejny duży fragment danych, o wielkości wyrównanej do rozmiaru bloku. Jeśli bufor posiada właściwą wielkość, uzyskuje się duże korzyści dzięki poprawie wydajności.

Poprzez jawne modyfikacje kodu programu możliwa jest implementacja buforowania w przestrzeni użytkownika. Faktycznie, postępuje tak wiele aplikacji o kluczowym znaczeniu. Jednakże większość programów używa popularnej *biblioteki typowych operacji wejścia i wyjścia* (ang. *standard I/O library*) (będącej elementem *standardowej biblioteki języka C*), która dostarcza solidnych rozwiązań dla buforowania w przestrzeni użytkownika.

Typowe operacje wejścia i wyjścia

Standardowa biblioteka języka C udostępnia bibliotekę typowych operacji wejścia i wyjścia (zwaną często w skrócie *stdio*), która dostarcza niezależnego od platformy rozwiązania dla buforowania w przestrzeni użytkownika. Biblioteka typowych operacji wejścia i wyjścia jest prosta w obsłudze, lecz jednocześnie posiada duże możliwości.

W przeciwieństwie do takich języków programowania jak FORTRAN, język C nie posiada wbudowanej obsługi ani też słów kluczowych, umożliwiających uzyskanie funkcjonalności na poziomie wyższym niż sterowanie działaniem, arytmetyka itd. — na pewno nie ma on wbudowanego wsparcia dla operacji wejścia i wyjścia. W trakcie rozwoju języka C, użytkownicy tworzyli typowe zestawy procedur umożliwiających uzyskanie podstawowej funkcjonalności, takie jak obsługa łańcuchów znakowych, procedury matematyczne, usługi związane z czasem i datą, operacje wejścia i wyjścia. Przez cały czas procedury te były rozwijane, a w momencie zatwierdzenia standardu ANSI C (C89) w 1989 roku zostały w końcu sformalizowane w postaci standardowej biblioteki języka C. Choć w kolejnych standardach C95 i C99 dodano kilka nowych interfejsów, jednak biblioteka typowych operacji wejścia i wyjścia nie została zbytnio zmieniona od momentu jej utworzenia w 1989 roku.

Pozostała część tego rozdziału poświęcona zostanie analizie takich operacji wejścia i wyjścia, które dotyczą plików i zostały zaimplementowane w standardowej bibliotece języka C, natomiast wykonywane są w przestrzeni użytkownika. Oznacza to, że omówione zostaną operacje otwierania, zamykania, czytania z plików i pisania do nich przy użyciu standardowej biblioteki języka C. Projektant oprogramowania, biorąc pod uwagę wymagania i zachowanie aplikacji, musi zawsze podejmować rozważną decyzję, czy powinny zostać użyte standardowe operacje wejścia i wyjścia, własne rozwiązanie buforowania w przestrzeni użytkownika lub też wywołania zwykłych funkcji systemowych.

W przypadku standardów języka C, niektóre szczegóły zawsze wymagają opracowania własnej implementacji, a te często są rozbudowywane poprzez dodawanie do nich nowych funkcji. W tym rozdziale, podobnie zresztą jak w pozostałej części książki, omówione zostaną interfejsy i zachowanie zgodne z implementacją *glibc* w nowoczesnym systemie linuksowym. Wskażemy również miejsca, w których zachowanie Linuksa odbiega od podstawowego standardu.

Wskaźniki do plików

Standardowe procedury wejścia i wyjścia nie przetwarzają bezpośrednio deskryptorów plików. Zamiast tego używają własnego, unikalnego deskryptora, znanego pod nazwą *wskaźnika do pliku* (ang. *file pointer*). Biblioteka języka C odwzorowuje wewnętrznie wskaźnik do pliku na jego deskryptor. Wskaźnik do pliku reprezentowany jest przez wskaźnik do typu `FILE`, którego definicja, zrealizowana przy użyciu konstrukcji językowej `typedef`, znajduje się w pliku nagłówkowym `<stdio.h>`.

W języku używanym dla opisu operacji wejścia i wyjścia plik otwarty zwany jest *strumieniem* (ang. *stream*). Strumienie mogą być otwierane do odczytu (*strumienie wejściowe*), zapisu (*strumienie wyjściowe*), lub obu operacji jednocześnie (*strumienie wejściowo-wyjściowe*).

Otwieranie plików

Pliki mogą zostać otwarte do odczytu lub zapisu za pomocą funkcji `fopen()`:

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

Funkcja ta otwiera w określonym trybie plik, którego nazwa podana jest w parametrze `path`, i kojarzy z nim nowy strumień.

Tryby

Parametr `mode` określa, w jaki sposób należy otworzyć dany plik. Parametr ten może być równy jednemu z poniższych łańcuchów:

`r`
: Otwiera plik do odczytu. Strumień ustawiony zostaje na początku pliku.

`r+`
: Otwiera plik do odczytu i zapisu. Strumień ustawiony zostaje na początku pliku.

`w`
: Otwiera plik do zapisu. Jeśli plik już istnieje, zostanie obcięty do długości zero. Jeśli plik nie istnieje, zostanie stworzony. Strumień ustawiony zostaje na początku pliku.

`w+`
: Otwiera plik do zapisu i odczytu. Jeśli plik już istnieje, zostanie obcięty do długości zero. Jeśli plik nie istnieje, zostanie stworzony. Strumień ustawiony zostaje na początku pliku.

`a`
: Otwiera plik do zapisu w trybie dopisywania. Jeśli plik nie istnieje, zostanie stworzony. Strumień ustawiony zostaje na końcu pliku. Wszystkie operacje zapisu będą powodować dopisywanie danych do tego pliku.

`a+`
: Otwiera plik do odczytu i zapisu w trybie dopisywania. Jeśli plik nie istnieje, zostanie stworzony. Strumień ustawiony zostaje na końcu pliku. Wszystkie operacje zapisu będą powodować dopisywanie danych do tego pliku.

Podany łańcuch znaków, określający dany tryb, może zawierać również literę `b`, choć w systemie Linux jest ona zawsze ignorowana. Niektóre systemy operacyjne odróżniają pliki tekstowe od binarnych, dlatego też litera `b` oznacza, że plik powinien zostać otwarty w trybie binarnym. Linux, podobnie jak wszystkie systemy zgodne ze standardem POSIX, obsługuje pliki tekstowe i binarne w taki sam sposób.

W przypadku sukcesu, funkcja `fopen()` zwraca poprawny wskaźnik do `FILE`. W przypadku błędu, zwraca `NULL` oraz odpowiednio ustawia `errno`.

Na przykład, poniższy kod otwiera plik */etc/manifest* do odczytu oraz kojarzy go ze strumieniem:

```
FILE *stream;

stream = fopen ("/etc/manifest", "r");
if (!stream)
   /* błąd */
```

Otwieranie strumienia poprzez deskryptor pliku

Funkcja `fdopen()` przekształca otwarty już deskryptor pliku `fd` w strumień:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

Dopuszczalne tryby są takie same jak w przypadku funkcji `fopen()`. Muszą być również kompatybilne z trybami, pierwotnie użytymi podczas otwierania deskryptora pliku. Tryby `w` i `w+` mogą zostać podane, lecz nie spowodują obcinania pliku. Strumień otrzyma taką pozycję w pliku, którą posiadał już dany deskryptor pliku.

Nie należy wykonywać już dalszych operacji wejścia i wyjścia bezpośrednio przy użyciu deskryptora pliku, gdy tylko zostanie on przekształcony w strumień. Jednak formalnie jest to dopuszczalne. Należy zauważyć, że deskryptor pliku nie zostaje powielony, lecz wyłącznie skojarzony z nowym strumieniem. Zamknięcie strumienia spowoduje również zamknięcie deskryptora pliku.

W przypadku sukcesu, funkcja `fdopen()` zwraca poprawny wskaźnik do pliku; w przypadku błędu, zwraca `NULL`.

Na przykład, poniższy kod otwiera plik */home/kidd/map.txt* przy użyciu funkcji systemowej `open()`, a następnie używa otrzymanego deskryptora pliku, aby utworzyć skojarzony z nim strumień:

```
FILE *stream;
int fd;

fd = open ("/home/kidd/map.txt", O_RDONLY);
if (fd == -1)
   /* błąd */

stream = fdopen (fd, "r");
if (!stream)
   /* błąd */
```

Zamykanie strumieni

Funkcja `fclose()` zamyka dany strumień:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Bufory, w których znajdują się dane niezapisane jeszcze na dysku, zostają opróżnione. W przypadku sukcesu, funkcja `fclose()` zwraca `0`. W przypadku błędu, zwraca `EOF` oraz odpowiednio ustawia zmienną `errno`.

Zamykanie wszystkich strumieni

Funkcja `fcloseall()` zamyka wszystkie strumienie skojarzone z aktualnym procesem, włącznie ze standardowym wejściem, standardowym wyjściem oraz standardowym wyjściem błędów:

```
#define _GNU_SOURCE

#include <stdio.h>

int fcloseall (void);
```

Wszystkie bufory danych dla strumieni zostają opróżnione. Funkcja ta zawsze zwraca `0`; jest ona specyficzna dla Linuksa.

Czytanie ze strumienia

Standardowa biblioteka języka C posiada zaimplementowanych wiele funkcji dla czytania z otwartego strumienia — zarówno powszechnie znanych, jak również takich, które są rzadko używane. W tym podrozdziale opisane zostaną trzy najbardziej powszechnie stosowane metody czytania danych: czytanie pojedynczych znaków, pojedynczych wierszy oraz danych binarnych. Aby móc czytać ze strumienia, należy go otworzyć do odczytu w odpowiednim trybie. Dopuszczalny jest każdy tryb za wyjątkiem trybów `w` lub `a`.

Czytanie pojedynczego znaku

Idealnym wzorcem projektowym jest często prosta operacja czytania pojedynczych znaków. Funkcja `fgetc()` może zostać użyta w celu odczytania jednego znaku ze strumienia:

```
#include <stdio.h>

int fgetc (FILE *stream);
```

Funkcja czyta kolejny znak ze strumienia `stream`, a zwracając go dokonuje jednocześnie rzutowania z typu `unsigned char` na typ `int`. Rzutowanie zastosowane zostaje w celu zachowania wystarczającego zakresu danych w przypadku komunikatu informującego o błędzie lub napotkanym końcu pliku. W takiej sytuacji następuje zwrócenie kodu `EOF`. Powrotna wartość funkcji `fgetc()` musi posiadać typ `int`. Przechowywanie tej wartości w zmiennej typu `char` jest często spotykanym i niebezpiecznym błędem.

Następujący przykład kodu odczytuje pojedynczy znak ze strumienia `stream`, sprawdza, czy nie pojawił się błąd oraz wyświetla wynik, rzutując go na typ `char`:

```
int c;

c = fgetc (stream);
if (c == EOF)
   /* błąd */
else
   printf ("c=%c\n", (char) c);
```

Strumień, wskazywany przez parametr `stream`, musi zostać wcześniej otwarty do odczytu.

Wycofywanie znaku

Standardowe operacje wejścia i wyjścia dostarczają funkcji pozwalającej na zwrócenie znaku z powrotem do strumienia. Dzięki temu możliwe jest „spojrzenie" do strumienia i zwrócenie znaku, jeśli okazało się, że nie jest on jednak potrzebny:

```
#include <stdio.h>

int ungetc (int c, FILE *stream);
```

Każde wywołanie funkcji wysyła znak `c` (po zrzutowaniu go na typ `unsigned char`) do strumienia `stream`. W przypadku sukcesu, funkcja zwraca wartość `c`; w przypadku błędu, zostaje zwrócony `EOF`. Kolejna próba odczytu ze strumienia `stream` zwróci znak `c`. Jeśli wycofywanych jest więcej znaków, kolejność ich pojawiania się w strumieniu jest odwrotna niż w przypadku czytania, to znaczy ostatni odczytany ze strumienia znak będzie wycofany do niego jako pierwszy. POSIX gwarantuje wycofanie z sukcesem tylko jednego znaku, bez potrzeby wykonywania

dodatkowych odczytów. W niektórych implementacjach możliwa jest wyłącznie jedna operacja wycofania. Linux zezwala na dowolną liczbę wycofań, o ile istnieją jeszcze wolne zasoby pamięci. Jedna operacja wycofania zawsze działa poprawnie.

Jeśli po wywołaniu funkcji `ungetc()`, lecz jeszcze przed wykonaniem operacji odczytu, nastąpi wywołanie funkcji szukającej (opisanej w podrozdziale Szukanie w strumieniu, znajdującym się w dalszej części tego rozdziału), spowoduje to, iż wycofane znaki zostaną odrzucone. Dotyczy to różnych wątków pojedynczego procesu, gdyż współdzielą one między sobą bufor z danymi.

Czytanie całego wiersza

Funkcja `fgets()` czyta łańcuch znaków z podanego strumienia:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

Funkcja ta czyta ze strumienia `stream` ciąg znaków o długości o 1 mniejszej, niż wynosi wartość parametru `size` i zapisuje go do bufora `str`. Po zakończeniu czytania do łańcucha dopisywany jest znak pusty (`\0`). Odczyt zostaje przerwany po osiągnięciu znaku `EOF` lub znaku nowej linii. Jeśli odczytano znak nowej linii, do bufora `str` dopisywany jest znak `\n`.

W przypadku sukcesu, funkcja zwraca wskaźnik do bufora `str`; w przypadku błędu, następuje zwrócenie `NULL`.

Na przykład:

```
char buf[LINE_MAX];

if (!fgets (buf, LINE_MAX, stream))
   /* błąd */
```

POSIX definiuje wartość stałej `LINE_MAX` w pliku nagłówkowym `<limits.h>`: jest to maksymalna długość wiersza wejściowego, który może być przetworzony przy użyciu interfejsów zgodnych ze standardem POSIX. Biblioteka języka C dla Linuksa nie ma takiego ograniczenia (wiersze mogą posiadać dowolną długość), lecz nie można tego zachowania powiązać z definicją `LINE_MAX`. W programach przenośnych należałoby używać stałej `LINE_MAX`, by mieć pewność, że będą działać poprawnie. Wartość tej stałej jest stosunkowo duża w Linuksie. W programach specyficznych dla Linuksa nie należy obawiać się żadnych problemów z ograniczeniem rozmiaru wiersza.

Czytanie dowolnych łańcuchów

Czytanie pojedynczych linii za pomocą funkcji `fgets()` jest często przydatne i jednocześnie irytujące. Czasami projektanci oprogramowania chcą używać innego ogranicznika niż znaku nowej linii. Zdarza się, że nie chcą w ogóle używać ograniczników. Tak naprawdę projektanci rzadko mają potrzebę, aby zapisywać znaki ogranicznika w buforze! Z perspektywy czasu decyzja o przechowywaniu znaku końca linii w zwróconym buforze rzadko wydaje się być poprawna.

Napisanie własnej wersji funkcji `fgets()`, która używa `fgetc()`, nie jest trudne. Na przykład, ten fragment kodu czyta `n - 1` bajtów ze strumienia `stream` do bufora `str`, a następnie dołącza znak `\0`:

```
char *s;
int c;

s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF)
   *s++ = c;
*s = '\0';
```

Powyższy kod może być uzupełniony tak, aby również przerywał odczyt w momencie napotkania ogranicznika, który podany zostaje w parametrze `d` (w tym przypadku nie może to być znak pusty):

```
char *s;
int c = 0;

s = str;
while (--n > 0 && (c = fgetc (stream)) != EOF && (*s++ = c) != d)
   ;
if (c == d)
   *--s = '\0';
else
   *s = '\0';
```

Ustawienie parametru `d` na wartość `\n` dostarczy zachowania podobnego do `fgets()`, za wyjątkiem tego, że znak nowej linii nie będzie zapisywany do bufora.

W porównaniu z innymi implementacjami funkcji `fgets()` ten wariant jest prawdopodobnie wolniejszy, gdyż wymaga wielokrotnego wykonywania funkcji `fgetc()`. Nie jest to jednak ten sam problem, który wcześniej został zasygnalizowany podczas użycia komendy *dd*! Choć zaprezentowany fragment kodu powoduje pewne dodatkowe obciążenie poprzez wielokrotne wywołanie funkcji `fgetc()`, nie obciąża jednak systemu przez nadmierne wywoływanie funkcji systemowych lub wykonywanie nieefektywnych operacji wejścia i wyjścia podczas użycia komendy `dd` z parametrem `bs=1` — co jest dużo poważniejszym problemem.

Czytanie danych binarnych

W przypadku niektórych aplikacji czytanie pojedynczych znaków lub linii nie jest wystarczające. Czasami wymagane jest czytanie lub zapisywanie złożonych danych binarnych, takich jak struktury języka C. Dlatego też biblioteka typowych operacji wejścia i wyjścia udostępnia funkcję `fread()`:

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

Wywołanie funkcji `fread()` spowoduje odczytanie pewnej liczby pakietów danych. Liczba ta przekazana jest w parametrze `nr`. Parametr `size` określa w bajtach rozmiar jednego pakietu. Dane czytane są ze strumienia `stream`, a zapisywane do bufora wskazywanego przez `buf`. Po przeczytaniu wskaźnik do pliku przesuwany jest o liczbę przeczytanych bajtów.

Funkcja zwraca liczbę przeczytanych pakietów (nie jest to liczba przeczytanych bajtów!). Funkcja informuje o błędzie lub końcu pliku (`EOF`) poprzez zwróconą wartość, mniejszą niż parametr `nr`. Niestety, bez użycia funkcji `ferror()` i `feof()` nie można określić, która z tych dwóch sytuacji miała miejsce (z tymi funkcjami można zapoznać się w kolejnym podrozdziale Błędy i koniec pliku).

Z powodu różnic w rozmiarach zmiennych, wyrównaniu, wypełnieniu i kolejności bajtów dane binarne zapisane za pomocą jednej aplikacji mogą nie zostać odczytane przez drugą aplikację lub nawet przez tę samą aplikację, ale działającą w innej maszynie.

Najprostszym przykładem użycia funkcji `fread()` jest kod, czytający dla danego strumienia pojedynczy pakiet danych z ciągu bajtów:

```
char buf[64];
size_t nr;

nr = fread (buf, sizeof(buf), 1, stream);
if (nr == 0)
   /* błąd */
```

Podczas analizy działania odpowiednika funkcji `fread()`, jakim jest `fwrite()`, przedstawione zostaną bardziej skomplikowane przykłady.

Pisanie do strumienia

Podobnie jak dla operacji czytania, standardowa biblioteka języka C definiuje wiele funkcji pozwalających pisać do otwartego strumienia. W tym podrozdziale opisane zostaną trzy najbardziej powszechnie stosowane metody zapisywania danych: zapisywanie pojedynczych znaków, łańcucha znaków oraz danych binarnych. Takie różne możliwości zapisu idealnie pasują do buforowanych operacji wejścia i wyjścia. Aby móc pisać do strumienia, należy otworzyć go do zapisu w odpowiednim trybie. Dopuszczalny jest każdy tryb, za wyjątkiem trybu `r`.

Zapisywanie pojedynczego znaku

Odpowiednikiem funkcji `fgetc()` jest funkcja `fputc()`:

```
#include <stdio.h>

int fputc (int c, FILE *stream);
```

Funkcja `fputc()` zapisuje bajt danych, przekazany w parametrze `c` (rzutowanym na typ `unsi`
`↪gned char`), do strumienia wskazywanego przez parametr `stream`. Po poprawnym wykonaniu funkcja zwraca znak `c`. W przeciwnym razie następuje zwrócenie `EOF` oraz odpowiednie ustawienie zmiennej `errno`.

Użycie funkcji jest proste:

```
if (fputc ('p', stream) == EOF)
   /* błąd */
```

Powyższy przykład przesyła znak `p` do strumienia `stream`, który musi być otwarty do zapisu.

Zapisywanie łańcucha znaków

Funkcja `fputs()` używana jest w celu zapisania całego łańcucha znaków do danego strumienia:

```
#include <stdio.h>

int fputs (const char *str, FILE *stream);
```

Problemy z wyrównaniem

We wszystkich architekturach maszynowych wymagane jest *wyrównanie danych*. Programiści mają skłonności do odbierania pamięci jako zwykłej tablicy bajtów. Nasze procesory jednak nie czytają z pamięci, ani nie zapisują do niej w paczkach jednobajtowych. Zamiast tego, procesory korzystają z pamięci, używając paczek o większych rozmiarach, na przykład: 2, 4, 8 czy 16 bajtów. Ponieważ przestrzeń adresowa dla każdego procesu rozpoczyna się od adresu 0, procesy mogą uzyskiwać dostęp do pamięci tylko od adresów, które są całkowitą wielokrotnością rozmiaru paczki.

Zgodnie z tym, zmienne języka C muszą być zapisywane oraz udostępniane od adresów wyrównanych do rozmiaru paczki. Ogólnie rzecz ujmując, zmienne są *wyrównane w sposób naturalny*. Dotyczy to wyrównania, które odpowiada rozmiarowi typu danych języka C. Na przykład, 32-bitowy typ całkowity wyrównany jest do granicy adresowej 4 bajtów. Inaczej mówiąc, zmienna typu `int` może być przechowywana w pamięci, poczynając od adresu, który jest dokładnie podzielny przez cztery.

Dostęp do niewyrównanych danych związany jest z wieloma problemami, które zależne są od architektury maszynowej. Niektóre procesory mogą uzyskać dostęp do takich danych, lecz wiąże się to z dużymi kosztami wydajnościowymi. Inne nie mają w ogóle dostępu do niewyrównanych danych, a próba dostania się do tego typu danych kończy się wygenerowaniem wyjątku sprzętowego. Co gorsze, niektóre procesory niejawnie pomijają mniej znaczące bity, aby wymusić wyrównanie adresu, co najprawdopodobniej powoduje niepożądane zachowanie aplikacji.

Dane są zwykle automatycznie wyrównywane przez kompilator i programista nie zauważa w ogóle tego problemu. Praca ze strukturami, jawne zarządzanie pamięcią, zapisywanie danych binarnych na dysk oraz komunikacja sieciowa mogą powodować jednak powstanie problemów z wyrównaniem. Programiści systemowi powinni zatem znać tego typu problemy!

Zagadnienie wyrównywania zostanie dokładniej omówione w rozdziale 8.

Wywołanie funkcji `fputs()` powoduje zapisanie całego łańcucha, wskazywanego przez parametr `str` i zakończonego znakiem pustym, do strumienia `stream`. W przypadku sukcesu, funkcja `fputs()` zwraca liczbę nieujemną. W przypadku błędu, następuje zwrócenie znaku `EOF`.

Poniższy przykład otwiera plik do zapisu w trybie dopisywania, zapisuje dany łańcuch do skojarzonego z plikiem strumienia, a następnie zamyka ten strumień:

```
FILE *stream;

stream = fopen ("journal.txt", "a");
if (!stream)
    /* błąd */

if (fputs ("Statek wykonany jest z drewna.\n", stream) == EOF)
    /* błąd */

if (fclose (stream) == EOF)
    /* błąd */
```

Zapisywanie danych binarnych

Jeśli w programie wymagane jest zapisywanie złożonych struktur danych, nie wystarczają funkcje, które zapisują pojedyncze znaki czy też łańcuchy znaków. Aby bezpośrednio zapisywać dane binarne, takie jak na przykład zmienne języka C, biblioteka typowych operacji wejścia i wyjścia udostępnia funkcję `fwrite()`:

```
#include <stdio.h>

size_t fwrite (void *buf, size_t size, size_t nr, FILE *stream);
```

Wywołanie funkcji `fwrite()` spowoduje zapisanie do strumienia `stream` określonej liczby pakietów danych. Liczba ta przekazana jest w parametrze `nr`. Parametr `size` określa rozmiar w bajtach jednego pakietu. Dane znajdują się w buforze, wskazywanym przez parametr `buf`, a zapisywane zostają do strumienia `stream`. Po przeprowadzeniu operacji zapisu wskaźnik do pliku przesuwany jest o liczbę zapisanych bajtów.

Funkcja zwraca liczbę zapisanych pakietów (nie jest to liczba zapisanych bajtów!). Jeśli zwrócona wartość jest mniejsza od `nr`, oznacza to błąd.

Przykładowy program używający buforowanych operacji wejścia i wyjścia

W niniejszym podrozdziale zostanie przedstawiony przykład kompletnego programu używającego wielu interfejsów, które do tej pory zostały omówione w tym rozdziale. W programie tym zostaje najpierw zdefiniowana struktura o nazwie `pirate`, a następnie zadeklarowane dwie zmienne tego typu. Program inicjalizuje jedną z tych zmiennych, a w dalszej kolejności zapisuje ją na dysk do pliku *data* przy użyciu strumienia wyjściowego. Następnie, przy użyciu kolejnego strumienia, dane z pliku *data* zostają bezpośrednio wczytane do innej kopii struktury `pirate`. Ostatecznie program wysyła zawartość struktury na standardowe wyjście:

```
#include <stdio.h>

int main (void)
{
   FILE *in, *out;
   struct pirate
   {
      char name[100]; /* imię i nazwisko */
      unsigned long booty; /* wartość łupu w funtach szterlingach */
      unsigned int beard_len; /* długość brody w centymetrach */
   } p, blackbeard = { "Edward Teach", 950, 122 };

   out = fopen ("data", "w");
   if (!out)
   {
      perror ("fopen");
      return 1;
   }

   if (!fwrite (&blackbeard, sizeof (struct pirate), 1, out))
   {
      perror ("fwrite");
      return 1;
   }
```

```
    if (fclose (out))
    {
       perror ("fclose");
       return 1;
    }

    in = fopen ("data", "r");
    if (!in)
    {
       perror ("fopen");
       return 1;
    }

    if (!fread (&p, sizeof (struct pirate), 1, in))
    {
       perror ("fread");
       return 1;
    }

    if (fclose (in))
    {
       perror ("fclose");
       return 1;
    }

    printf ("Imię i nazwisko = \"%s\", wartość łupu = %lu funtów szterlingów, długość
brody = %u
     cm\n", p.name, p.booty, p.beard_len);

    return 0;
}
```

Po uruchomieniu tego programu uzyskuje się następujący wynik:

```
Imię i nazwisko = "Edward Teach", wartość łupu = 950 funtów szterlingów, długość brody
= 122 cm
```

Ważne jest, aby pamiętać, że z powodu różnic w rozmiarach zmiennych, wyrównaniu itd. dane binarne zapisane za pomocą jednej aplikacji mogą nie zostać odczytane przez drugą aplikację. Oznacza to, że inna aplikacja (a nawet ta sama aplikacja, lecz działająca w kolejnej maszynie) może niepoprawnie odczytać dane zapisane za pomocą funkcji `fwrite()`. Jeśli dla powyższego przykładu zmienił się rozmiar typu `unsigned long` lub liczba dopełniających bajtów, może to spowodować powstanie wspomnianego wcześniej problemu. Niezmienność tych czynników gwarantowana jest jedynie dla aplikacji działających w maszynach o takim samym typie i posiadających taki sam interfejs binarny aplikacji (ABI).

Szukanie w strumieniu

Często przydatna jest możliwość zmiany aktualnej pozycji w strumieniu. Być może aplikacja czyta ze skomplikowanego pliku, posiadającego strukturę rekordów, lub musi wykonać skok do innej pozycji. Z drugiej strony, być może wymagane jest ustawienie pozycji w strumieniu na początku pliku. Bez względu na to co jest powodem zmiany pozycji w strumieniu, biblioteka typowych operacji wejścia i wyjścia dostarcza zestawu interfejsów, których funkcjonalność równoważna jest działaniu funkcji systemowej `lseek()` (omówionej w rozdziale 2.). Funkcja `fseek()` to najbardziej znany interfejs biblioteki typowych operacji wejścia i wyjścia z grupy interfejsów, obsługujących szukanie. Pozwala ona na modyfikowanie położenia w strumieniu `stream` przy użyciu dodatkowych parametrów `offset` i `whence`:

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

Jeśli parametr `whence` równy jest `SEEK_SET`, pozycja w pliku zostanie ustawiona na wartość równą parametrowi `offset`. Jeśli parametr `whence` równy jest `SEEK_CUR`, pozycja w pliku zostanie ustawiona na wartość równą sumie aktualnej pozycji oraz parametru `offset`. Jeśli parametr `whence` wynosi `SEEK_END`, pozycja w pliku zostanie ustawiona na wartość równą sumie pozycji końca pliku oraz parametru `offset`.

W przypadku poprawnego wykonania, funkcja `fseek()` zwraca `0`, zeruje wskaźnik `EOF` oraz przywraca stan sprzed ewentualnego wywołania funkcji `ungetc()`. W przypadku błędu, zwraca `-1` oraz odpowiednio ustawia zmienną `errno`. Najczęściej spotykanymi błędami są: nieprawidłowy strumień (`EBADF`) oraz nieprawidłowa wartość parametru `whence` (`EINVAL`).

Jako alternatywę dla funkcji `fseek()` biblioteka typowych operacji wejścia i wyjścia udostępnia również funkcję `fsetpos()`:

```
#include <stdio.h>

int fsetpos (FILE *stream, fpos_t *pos);
```

Funkcja ta ustawia pozycję w strumieniu na wartość `pos`. Działa jak funkcja `fseek()` z parametrem `whence` równym `SEEK_SET`. W przypadku sukcesu, funkcja zwraca `0`. W przypadku błędu, zwraca `-1` oraz odpowiednio ustawia zmienną `errno`. Ta funkcja (razem z jej odpowiednikiem — funkcją `fgetpos()`, która zostanie za chwilę omówiona) udostępniana jest jedynie dla innych (nieuniksowych) platform, które posiadają złożone typy danych, reprezentujące pozycję w strumieniu. Użycie funkcji `fsetpos()`jest wtedy jedynym sposobem na ustawienie pozycji w strumieniu na określoną wartość, ponieważ wykorzystanie typu języka C `long long` będzie prawdopodobnie niewystarczające. Ten interfejs nie musi być używany w aplikacjach specyficznych dla Linuksa, choć może zostać wykorzystany, jeśli należy zapewnić maksymalną przenośność aplikacji.

Biblioteka typowych operacji wejścia i wyjścia udostępnia także uproszczoną funkcję o nazwie `rewind()`:

```
#include <stdio.h>

void rewind (FILE *stream);
```

Poniższe wywołanie funkcji `rewind()` spowoduje ustawienie pozycji w pliku na początek strumienia `stream`:

```
rewind(stream);
```

To wywołanie jest odpowiednikiem następującego wywołania funkcji `fseek()` (za wyjątkiem tego, że `rewind()` zeruje wskaźnik błędu):

```
fseek (stream, 0, SEEK_SET);
```

Należy zwrócić uwagę na to, że funkcja `rewind()` nie zwraca żadnej wartości, dlatego też nie może bezpośrednio poinformować o wystąpieniu błędu. Procedury, które wymagają informacji o błędach, powinny zerować zmienną `errno` przed wywołaniem funkcji `rewind()`, a następnie po jej użyciu sprawdzać, czy `errno` nie zmieniło swojej wartości. Na przykład:

```
errno = 0;
rewind (stream);
if (errno)
    /* błąd */
```

Otrzymywanie informacji o aktualnym położeniu w strumieniu

W przeciwieństwie do `lseek()` funkcja `fseek()` nie zwraca aktualnego położenia w pliku. Dla tego celu udostępniony jest inny interfejs. Funkcja `ftell()` zwraca aktualne położenie w strumieniu `stream`:

```
#include <stdio.h>

long ftell (FILE *stream);
```

W przypadku błędu, funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno`.

Jako alternatywę dla funkcji `ftell()` biblioteka typowych operacji wejścia i wyjścia udostępnia również funkcję `fgetpos()`:

```
#include <stdioh.h>

int fgetpos (FILE *stream, fpos_t *pos);
```

W przypadku sukcesu, funkcja zwraca `0` oraz zapisuje w parametrze `pos` wartość aktualnego położenia w strumieniu `stream`. W przypadku błędu, zwraca `-1` oraz odpowiednio ustawia zmienną `errno`. Podobnie jak `fsetpos()` funkcja `fgetpos()` udostępniana jest wyłącznie dla platform nielinuksowych, które posiadają złożone typy danych, reprezentujące położenie w strumieniu.

Opróżnianie strumienia

Biblioteka typowych operacji wejścia i wyjścia dostarcza interfejsu, który pozwala na zapisanie bufora z przestrzeni użytkownika do jądra, zapewniając, że wszystkie dane wprowadzone do strumienia zostaną opróżnione z niego poprzez wywołanie funkcji `write()`. Funkcja `fflush()` umożliwia wykonanie poniższej operacji:

```
#include <stdio.h>

int fflush (FILE *stream);
```

Podczas wywołania tej funkcji wszystkie niezapisane dane ze strumienia `stream` zostają przerzucone do jądra. Jeśli parametr `stream` równy jest `NULL`, *każdy* z otwartych strumieni dla danego procesu zostanie opróżniony. W przypadku sukcesu, funkcja zwraca `0`. W przypadku błędu, zwraca `EOF` oraz odpowiednio ustawia zmienną `errno`.

Aby zrozumieć rezultaty działania funkcji `fflush()`, należy najpierw pojąć, na czym polega różnica między buforem obsługiwanym przez bibliotekę języka C, a buforowaniem przeprowadzanym przez jądro. Wszystkie funkcje opisane w tym rozdziale używają bufora, który obsługiwany jest przez bibliotekę języka C i znajduje się w przestrzeni użytkownika, a nie w przestrzeni jądra. Tu właśnie zachodzi poprawa wydajności — aplikacja pozostaje w przestrzeni użytkownika, działając wyłącznie w jego kodzie i nie wywołując funkcji systemowych. Funkcja systemowa zostaje wywołana tylko wtedy, gdy należy uzyskać dostęp do dysku lub innego medium.

Funkcja `fflush()` jedynie zapisuje dane, buforowane w przestrzeni użytkownika, do bufora jądra. Wynik jej działania jest taki sam, jak w przypadku, gdyby nie istniało buforowanie w przestrzeni użytkownika, a funkcja `write()` była bezpośrednio używana. Wywołanie funkcji `fflush()` nie gwarantuje, że dane zostaną fizycznie zapisane na jakimś medium — w tym celu należy użyć

funkcji `fsync()` (opisanej w podrozdziale Zsynchronizowane operacje wejścia i wyjścia, znajdującym się w rozdziale 2.). Zalecany jest następujący sposób działania: wywołać funkcję `fflush()`, a natychmiast po niej wywołać `fsync()`: oznacza to, że najpierw należy zapewnić wysłanie danych z bufora w przestrzeni użytkownika do jądra, a następnie zapewnić zapisanie bufora z jądra na dysk.

Błędy i koniec pliku

Niektóre ze standardowych interfejsów wejścia i wyjścia, takie jak funkcja `fread()`, w ograniczony sposób informują procedurę wywołującą o niepowodzeniu wykonania, gdyż nie posiadają mechanizmu, który pozwala rozpoznać różnicę pomiędzy błędem, a osiągnięciem końca pliku. Dla tej funkcji i również innych przypadków, użyteczne może być sprawdzenie stanu danego strumienia, by ustalić, czy pojawił się błąd, czy też osiągnięto koniec pliku. W tym celu biblioteka typowych operacji wejścia i wyjścia dostarcza dwóch interfejsów. Funkcja `ferror()` sprawdza, czy dla strumienia `stream` został ustawiony wskaźnik błędu:

```
include <stdio.h>

int ferror (FILE *stream);
```

Wskaźnik błędu zostaje ustawiony przez inne interfejsy wejścia i wyjścia jako odpowiedź na pojawienie się błędu. Jeśli wskaźnik jest ustawiony, funkcja zwraca wartość niezerową; w przeciwnym razie zwraca zero.

Funkcja `feof()` sprawdza, czy dla strumienia `stream` został ustawiony znacznik końca pliku `EOF`:

```
include <stdio.h>

int feof (FILE *stream);
```

Znacznik `EOF` zostaje ustawiony przez inne typowe interfejsy wejścia i wyjścia, gdy osiągnięto koniec pliku. Jeśli znacznik jest ustawiony, funkcja zwraca wartość niezerową; w przeciwnym razie zwraca zero.

Funkcja `clearerr()` zeruje znaczniki błędu i końca pliku dla strumienia `stream`:

```
#include <stdio.h>

void clearerr (FILE *stream);
```

Funkcja nie zwraca żadnej wartości powrotnej i nie może być wykonana niepoprawnie (nie istnieje metoda pozwalająca sprawdzić, czy dostarczono niepoprawny identyfikator strumienia). Funkcja `clearerr()` powinna być wywołana jedynie wtedy, gdy znaczniki błędu i `EOF` zostały już sprawdzone, ponieważ po jej wywołaniu zostaną one wyzerowane i nie będzie można odzyskać ich poprzednich wartości. Na przykład:

```
/* 'f' oznacza identyfikator poprawnego strumienia */

if (ferror (f))
   printf ("Wystąpił błąd w strumieniu f!\n");

if (feof (f))
   printf ("Osiągnięto koniec pliku dla strumienia f!\n");

clearerr (f);
```

Otrzymywanie skojarzonego deskryptora pliku

Czasami warto uzyskać dostęp do deskryptora pliku, który związany jest z danym strumieniem. Na przykład, gdy dla strumienia nie istnieje odpowiednia funkcja z biblioteki typowych operacji wejścia i wyjścia, mogłoby okazać się użyteczne wywołanie dla niego funkcji systemowej przy użyciu związanego z nim deskryptora pliku. By otrzymać deskryptor pliku, związany ze strumieniem, należy wywołać funkcję `fileno()`:

```
#include <stdio.h>

int fileno (FILE *stream);
```

W przypadku sukcesu, funkcja `fileno()` zwraca deskryptor pliku związany ze strumieniem `stream`. W przypadku błędu, zwraca `-1`. Może się to zdarzyć jedynie wtedy, gdy dany identyfikator strumienia będzie niepoprawny. W tym przypadku zmienna `errno` zostanie ustawiona na wartość `EBADF`.

Łączenie wywołań funkcji z biblioteki typowych operacji wejścia i wyjścia z wywołaniami funkcji systemowych nie jest zwykle zalecane. Programiści muszą zwracać uwagę na poprawne działanie aplikacji podczas używania funkcji `fileno()`. Szczególnie ważne jest opróżnienie strumienia, zanim nastąpi użycie związanego z nim deskryptora pliku. Praktycznie nigdy nie należy łączyć ze sobą rzeczywistych operacji wejścia i wyjścia.

Parametry buforowania

Standardowa biblioteka wejścia i wyjścia implementuje trzy rodzaje buforowania w przestrzeni użytkownika oraz dostarcza projektantom interfejs pozwalający na wybór rodzaju i rozmiaru bufora. Różne rodzaje buforowania w przestrzeni użytkownika przeznaczone są do różnych celów i najlepiej sprawdzają się w odmiennych sytuacjach. Oto one:

Brak buforowania
: Nie jest używane żadne buforowanie w przestrzeni użytkownika. Dane przekazywane są bezpośrednio do jądra. Ponieważ to zachowanie jest przeciwieństwem buforowania w przestrzeni użytkownika, nie jest ono powszechnie stosowane. Standardowe wyjście błędów domyślnie pracuje w trybie braku buforowania.

Buforowanie wierszowe
: Buforowanie wykonywane jest w trybie wierszowym. Po każdym znaku nowej linii bufor zostaje przesłany do jądra. Buforowanie wierszowe ma sens dla strumieni, których zawartość wysyłana jest na ekran. Zgodnie z tym, jest to domyślny sposób buforowania dla terminali (standardowe wyjście jest domyślnie buforowane w trybie wierszowym).

Buforowanie blokowe
: Buforowanie wykonywane jest w trybie blokowym. To rodzaj buforowania omawianego na początku tego rozdziału. Jest ono idealne dla pracy z plikami. Domyślnie, wszystkie strumienie związane z plikami są buforowane w trybie blokowym. Biblioteka typowych operacji wejścia i wyjścia używa określenia *buforowanie pełne* jako zamiennika dla buforowania blokowego.

Domyślnie wybrany tryb buforowania zazwyczaj działa poprawnie i optymalnie. Jednakże biblioteka typowych operacji wejścia i wyjścia dostarcza interfejs, który pozwala na ustalenie rodzaju zastosowanego buforowania:

```
#include <stdio.h>

int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

Funkcja `setvbuf()` ustala dla strumienia `stream` rodzaj buforowania na tryb, przekazany w parametrze `mode`. Parametr ten musi być ustawiony na jedną z trzech poniższych wartości:

`_IONBF`
: Oznacza brak buforowania.

`_IOLBF`
: Oznacza buforowanie wierszowe.

`_IOFBF`
: Oznacza buforowanie blokowe (pełne).

Za wyjątkiem wartości `_IONBF`, dla której parametry `buf` i `size` są ignorowane, `buf` wskazuje na obszar pamięci o rozmiarze `size`, który zostanie ustawiony jako domyślny bufor dla standardowych operacji wejścia i wyjścia dla danego strumienia `stream`. Jeśli parametr `buf` wynosi `NULL`, bufor przydzielony zostanie automatycznie przez bibliotekę *glibc*.

Funkcja `setvbuf()` musi zostać wywołana już po otwarciu strumienia, ale przed rozpoczęciem związanych z nim innych operacji. W przypadku sukcesu, funkcja zwraca `0`. W przypadku błędu, zwraca wartość niezerową.

Bufor, przekazany do funkcji, musi istnieć, gdy strumień zostaje zamknięty. Powszechnie popełnianym błędem jest deklarowanie bufora będącego automatyczną zmienną, posiadającą pewien zakres dostępności, która zostaje usunięta przez aplikację przed zamknięciem strumienia. Należy zwrócić szczególną uwagę, by bufor nie był lokalną zmienną w funkcji `main()`, gdyż może to spowodować niepowodzenie podczas jawnego zamykania strumienia. Na przykład, poniższy kod jest błędny:

```
#include <stdio.h>

int main (void)
{
   char buf[BUFSIZ];

   /* ustaw stdout w tryb buforowania blokowego z buforem o rozmiarze BUFSIZ */
   setvbuf (stdout, buf, _IOFBF, BUFSIZ);
   printf ("Hej ho!\n");

   return 0;
}
```

Błąd może zostać naprawiony przez jawne zamknięcie strumienia, zanim zmienna `buf`, reprezentująca bufor, przestanie być dostępna, lub poprzez zadeklarowanie `buf` jako zmiennej globalnej.

Zwykle projektanci oprogramowania nie muszą zajmować się buforowaniem strumienia. Za wyjątkiem standardowego wyjścia błędów, terminale działają w trybie buforowania wierszowego i ma to sens. Pliki posiadają buforowanie blokowe i to również jest sensowne. Domyślny

rozmiar bufora dla buforowania blokowego wynosi `BUFSIZ` i jest zdefiniowany w pliku nagłówkowym `<stdio.h>`. Najczęściej jest on optymalną wartością dla aplikacji (wielokrotnością typowego rozmiaru bloku o dużej wartości).

Bezpieczeństwo wątków

Wątki są jednostkami aktywności wewnątrz procesu. Można sobie je wyobrazić jako dużą liczbę procesów dzielących tą samą przestrzeń adresową. Wątki mogą być uruchamiane w dowolnym momencie, jak również mogą nadpisywać współdzielone dane, dopóki nie zastosuje się środków bezpieczeństwa, dzięki którym dostęp do danych zostanie zsynchronizowany lub staną się one dostępne tylko dla poszczególnych wątków. Systemy operacyjne, które udostępniają wątki, dostarczają mechanizmów blokowania (konstrukcji programowych, zapewniających wzajemne wykluczenie), dzięki którym wątki nie przeszkadzają sobie wzajemnie. Biblioteka typowych operacji wejścia i wyjścia używa takich mechanizmów. Mimo tego nie zawsze są one wystarczające. Na przykład, czasami konieczne jest zablokowanie pewnej grupy wywołań funkcji poprzez takie powiększenie regionu krytycznego (fragmentu kodu, do którego ma dostęp tylko jeden wątek), aby zamiast jednej obejmował kilka operacji wejścia i wyjścia. W przypadku innych sytuacji być może konieczne będzie całkowite wyeliminowanie blokowania, w celu uzyskania poprawy wydajności[1]. Oba te tematy zostaną przeanalizowane w tym podrozdziale.

Standardowe funkcje wejścia i wyjścia w zasadzie obsługują wątki w bezpieczny sposób. Wewnętrznie każdemu otwartemu strumieniowi przypisany zostaje mechanizm blokady, licznik blokady oraz proces właścicielski. Przed wykonaniem dowolnego żądania, związanego z operacjami wejścia i wyjścia, każdy wątek musi założyć blokadę oraz stać się procesem właścicielskim. Dwa lub więcej wątków, korzystających z tego samego strumienia, nie może na przemian używać standardowych operacji wejścia i wyjścia, dlatego też w kontekście pojedynczych wywołań funkcji, standardowe operacje wejścia i wyjścia są niepodzielne (atomowe).

W praktyce wiele aplikacji wymaga większej niepodzielności od tej, która istnieje tylko na poziomie wywołań funkcji. Na przykład, jeśli wiele wątków wysyła żądania zapisu, wówczas aplikacja może wymagać, aby cała grupa operacji zapisu zakończyła się bez przerywania, mimo że pojedyncze operacje na pewno nie będą przemiennie wykonywać się dla różnych wątków i w rezultacie zniekształcać dane w strumieniu wyjściowym. By umożliwić takie zachowanie, standardowe operacje wejścia i wyjścia udostępniają całą rodzinę funkcji pozwalających indywidualnie zmieniać ustawienia blokad związanych ze strumieniami.

Nieautomatyczne blokowanie plików

Funkcja `flockfile()` czeka, dopóki strumień `stream` nie utraci blokady, a wówczas zakłada swoją blokadę, zwiększa jej licznik i wraca do procedury wywołującej, stając się wątkiem właścicielskim dla strumienia:

```
#include <stdio.h>

void flockfile (FILE *stream);
```

[1] Eliminacja blokowania powoduje zazwyczaj powstanie problemów. Jednak w przypadku niektórych programów można jawnie implementować własny sposób użycia wątków, pozwalający na obsługę wszystkich operacji wejścia i wyjścia przez jeden wątek. W przypadku tego rozwiązania nie istnieją dodatkowe koszty blokowania.

Funkcja `funlockfile()` zmniejsza licznik blokady, związany ze strumieniem `stream`:

```
#include <stdio.h>

void funlockfile (FILE *stream);
```

Jeśli licznik blokady osiągnie wartość zero, następuje zrzeczenie się praw własności do strumienia przez aktualny wątek. W tym momencie inny wątek może założyć swoją blokadę.

Funkcje te mogą być zagnieżdżone. Oznacza to, że pojedynczy wątek może wielokrotnie wywołać funkcję `flockfile()`, a strumień zostanie odblokowany dopiero wtedy, gdy proces tyle samo razy wywoła funkcję `funlockfile()`.

Funkcja `ftrylockfile()` jest nieblokującą wersją funkcji `flockfile()`:

```
#include <stdio.h>

int ftrylockfile (FILE *stream);
```

Jeśli w momencie wywołania funkcji strumień `stream` jest zablokowany, nie wykonuje ona niczego i natychmiast powraca z wartością niezerową. Jeśli strumień `stream` nie jest zablokowany, funkcja zakłada blokadę, zwiększa licznik blokady, staje się wątkiem właścicielskim dla strumienia i ostatecznie zwraca zero.

Rozważmy następujący przykład:

```
flockfile (stream);

fputs ("Lista skarbów:\n", stream);
fputs (" (1) 500 złotych monet\n", stream);
fputs (" (2) Wspaniale zdobiona zastawa stołowa\n", stream);

funlockfile (stream);
```

Choć pojedyncze wywołania funkcji `fputs()` nigdy nie mogłyby współzawodniczyć ze sobą (na przykład, „Lista skarbów" nie mogłaby być przerwana niczym innym), inna standardowa operacja wejścia i wyjścia, pochodząca z odmiennego wątku, mogłaby wykonać się pomiędzy dwiema kolejnymi funkcjami `fputs()`. Idealnie byłoby, gdyby aplikacja została zaprojektowana w taki sposób, by wiele wątków nie mogło wykonywać operacji wejścia i wyjścia dla tego samego strumienia. Jeśli aplikacja wymaga takiego zachowania, a programista potrzebuje niepodzielnego obszaru obejmującego więcej niż tylko wywołanie jednej funkcji, `flockfile()` i funkcje z nią pokrewne mogą zaoszczędzić wiele wysiłku.

Nieblokowane operacje na strumieniu

Istnieje jeszcze inny powód, aby wykonywać nieautomatyczną blokadę strumienia. Dzięki zastosowaniu bardzo dokładnej i precyzyjnej kontroli blokowania, która może być zrealizowana wyłącznie przez programistę aplikacji, możliwe jest zminimalizowanie kosztów blokowania oraz poprawienie wydajności. W tym celu Linux dostarcza zestawu funkcji, które nie przeprowadzają żadnego blokowania i podobne są do zwykłych interfejsów dla typowych operacji wejścia i wyjścia. Faktycznie są one nieblokującymi odpowiednikami zwykłych funkcji wejścia i wyjścia:

```
#define _GNU_SOURCE

#include <stdio.h>

int fgetc_unlocked (FILE *stream);
```

```
char *fgets_unlocked (char *str, int size, FILE *stream);
size_t fread_unlocked (void *buf, size_t size, size_t nr, FILE *stream);
int fputc_unlocked (int c, FILE *stream);
int fputs_unlocked (const char *str, FILE *stream);
size_t fwrite_unlocked (void *buf, size_t size, size_t nr, FILE *stream);
int fflush_unlocked (FILE *stream);
int feof_unlocked (FILE *stream);
int ferror_unlocked (FILE *stream);
int fileno_unlocked (FILE *stream);
void clearerr_unlocked (FILE *stream);
```

Funkcje te zachowują się tak samo jak ich blokujące odpowiedniki, za wyjątkiem tego, że nie sprawdzają, ani nie zakładają blokady związanej z danym strumieniem `stream`. Gdy pojawia się wymóg blokowania, programista staje się odpowiedzialny za ręczne założenie, a następnie usunięcie blokady.

Mimo że standard POSIX definiuje pewne nieblokujące warianty standardowych funkcji wejścia i wyjścia, jednak żadna z powyższych funkcji nie została przez niego zdefiniowana. Wszystkie one są specyficzne dla Linuksa, chociaż istnieje wsparcie dla ograniczonego podzbioru tych funkcji również ze strony innych systemów uniksowych.

Krytyczna analiza biblioteki typowych operacji wejścia i wyjścia

Mimo że biblioteka typowych operacji wejścia i wyjścia jest szeroko używana, eksperci wskazują na pewne jej wady. Niektóre funkcje, takie jak `fgets()`, są czasami niewystarczające. Inne, jak na przykład `gets()`, są tak nieprzyjazne, że zostały praktycznie usunięte z definicji standardów.

Największą dolegliwością związaną z biblioteką typowych operacji wejścia i wyjścia, jest pogorszenie sprawności, spowodowane podwójnym kopiowaniem. Podczas czytania, biblioteka typowych operacji wejścia i wyjścia wywołuje funkcję systemową `read()`, kopiując dane z jądra do bufora biblioteki. Gdy aplikacja wykonuje następnie operację czytania poprzez bibliotekę typowych operacji wejścia i wyjścia — na przykład, używając funkcji `fgetc()` — dane zostają ponownie skopiowane, tym razem z bufora biblioteki do bufora aplikacji. Operacje zapisu funkcjonują w odwrotny sposób: dane skopiowane zostają po raz pierwszy z bufora aplikacji do bufora biblioteki, a następnie po raz drugi z bufora biblioteki do jądra poprzez użycie funkcji `write()`.

Alternatywna implementacja mogłaby zabezpieczać przed podwójnym kopiowaniem dzięki zwracaniu po każdej operacji czytania wskaźnika do bufora, znajdującego się w bibliotece typowych operacji wejścia i wyjścia. Dane mogłyby być następnie bezpośrednio czytane z bufora biblioteki, bez potrzeby tworzenia dodatkowej kopii. W przypadku, gdyby aplikacja potrzebowała danych w swoim lokalnym buforze — by je modyfikować — mogłaby zawsze wykonać ręczne kopiowanie. Implementacja ta dostarczałaby „wolnodostępnego" interfejsu, dzięki któremu aplikacje mogłyby sygnalizować zakończenie obsługi danego fragmentu danych w buforze odczytu.

Operacje zapisu byłyby trochę bardziej skomplikowane, lecz w dalszym ciągu można by uniknąć podwójnego kopiowania. Podczas wykonywania operacji zapisu implementacja zapamiętywałaby wskaźnik. W momencie gotowości do operacji wyrzucenia danych do jądra, implementacja mogłaby przetworzyć listę wszystkich zapamiętanych wskaźników, zapisując wskazywane

przez nie dane. Może to być zrealizowane za pomocą *rozproszonych operacji wejścia i wyjścia* (ang. *scatter-gather I/O*), a dokładnie przy użyciu tylko jednej funkcji `writev()` (rozproszone operacje wejścia i wyjścia zostaną omówione w następnym rozdziale).

Istnieją biblioteki o wysokim stopniu zoptymalizowania, służące do buforowania w przestrzeni użytkownika i rozwiązujące problem podwójnego kopiowania za pomocą implementacji podobnych do wyżej wspomnianej. Zdarza się także, że projektanci oprogramowania implementują swoje własne rozwiązania buforowania w przestrzeni użytkownika. Pomimo różnych możliwości, biblioteka typowych operacji wejścia i wyjścia jest wciąż popularna.

Zakończenie

Biblioteka typowych operacji wejścia i wyjścia, będąca częścią standardowej biblioteki języka C, przeznaczona jest do buforowania danych w przestrzeni użytkownika. Mimo kilku wad posiada duże możliwości i jest bardzo popularna. Wielu programistów języka C zna tylko tę bibliotekę. Na pewno dla terminalowych operacji wejścia i wyjścia, gdzie buforowanie oparte na trybie wierszowym jest idealne, jest ona jedynym sensownym rozwiązaniem. Czy ktoś kiedykolwiek wywoływał bezpośrednio funkcję `write()`, by wysyłać znaki do standardowego wyjścia?

Użycie biblioteki typowych operacji wejścia i wyjścia — oraz, w tym przypadku, ogólnie buforowanie w przestrzeni użytkownika — ma sens, gdy jeden z poniższych przypadków zostaje spełniony:

- Można zminimalizować koszty poprzez zastąpienie wielu wywołań funkcji systemowych kilkoma wywołaniami innych funkcji z poziomu użytkownika.
- Wydajność działania jest krytycznym parametrem, a wszystkie operacje wejścia i wyjścia mają wykonywać się przy użyciu pakietów dopasowanych i wyrównanych do rozmiaru bloku.
- Sposób dostępu do danych oparty jest na przetwarzaniu znaków i wierszy, a jednocześnie wymagane są takie interfejsy, dzięki którym można łatwo uzyskać ten dostęp, bez potrzeby wykonywania dodatkowych funkcji systemowych.
- Zamiast niskopoziomowych funkcji systemowych Linuksa preferowane są raczej interfejsy wyższego poziomu.

Największa elastyczność istnieje jednak w przypadku bezpośredniego używania funkcji systemowych Linuksa. W następnym rozdziale omówione zostaną zaawansowane metody przeprowadzania operacji wejścia i wyjścia oraz związane z nimi funkcje systemowe.

ROZDZIAŁ 4.

Zaawansowane operacje plikowe wejścia i wyjścia

W rozdziale 2. przeanalizowano funkcje systemowe dotyczące przeprowadzania podstawowych operacji wejścia i wyjścia w systemie Linux. Funkcje te tworzą nie tylko podstawy operacji plikowych wejścia i wyjścia, ale również są fundamentem, na którym oparta jest właściwie cała komunikacja w Linuksie. W rozdziale 3. wyjaśniono, w jaki sposób buforowanie w przestrzeni użytkownika zastępuje funkcje systemowe, przeznaczone dla wykonywania podstawowych operacji wejścia i wyjścia, a także przeanalizowano specyficzne rozwiązanie buforowania w przestrzeni użytkownika, jakim jest biblioteka typowych operacji wejścia i wyjścia dla języka C. W tym rozdziale zostaną poddane analizie bardziej zaawansowane funkcje systemowe dotyczące operacji wejścia i wyjścia, które udostępniane są przez system Linux:

Rozproszone operacje wejścia i wyjścia
: Pozwalają na wykonanie jednego wywołania funkcji w celu jednoczesnego odczytu lub zapisu danych z wielu buforów; są użyteczne w przypadku połączenia pól lub różnych struktur danych, by zrealizować jedną transakcję wejścia i wyjścia.

Epoll
: Ulepsza działanie funkcji systemowych `poll()` i `select()` omówionych w rozdziale 2.; rozwiązanie to jest użyteczne, gdy pojedynczy program musi przetworzyć setki deskryptorów plików.

Operacje wejścia i wyjścia odwzorowane w pamięci
: Odwzorowują plik na obszar pamięci, pozwalając, aby operacje plikowe wejścia i wyjścia były realizowane tylko poprzez proste manipulowanie pamięcią; są użyteczne dla pewnych wzorców programowania związanych z wejściem i wyjściem.

Porada dla plikowych operacji wejścia i wyjścia
: Umożliwia dostarczenie jądru wskazówek na temat scenariuszy użycia procesu; może przyczyniać się do poprawy wydajności związanej z operacjami wejścia i wyjścia.

Asynchroniczne operacje wejścia i wyjścia
: Umożliwiają wysyłanie przez proces żądań przeprowadzenia operacji wejścia i wyjścia bez oczekiwania na ich zakończenie; są użyteczne w przypadku przetwarzania dużej liczby operacji wejścia i wyjścia bez potrzeby użycia wątków.

Rozdział ten zostanie zakończony analizą czynników wydajnościowych oraz podsystemów jądra, odpowiedzialnych za operacje wejścia i wyjścia.

Rozproszone operacje wejścia i wyjścia

Rozproszone operacje wejścia i wyjścia pozwalają na przeprowadzenie zapisu danych z wektora buforów do jednego strumienia przy użyciu tylko pojedynczego wywołania funkcji systemowej lub pozwalają na odczyt danych z jednego strumienia do wektora buforów. Ten rodzaj operacji wejścia i wyjścia posiada taką nazwę, ponieważ dane mogą być *rozproszone* po wektorze buforów, aby następnie być z niego pobranymi i ponownie skonsolidowanymi. *Wektorowe operacje wejścia i wyjścia* to inna nazwa dla tego rodzaju metody obsługi operacji wejścia i wyjścia. W przeciwieństwie do tej metody standardowe funkcje systemowe dla odczytu i zapisu, które zostały omówione w rozdziale 2., pozwalają na wykonywanie *liniowych operacji wejścia i wyjścia*.

Rozproszone operacje wejścia i wyjścia posiadają kilka zalet w porównaniu z liniowymi operacjami wejścia i wyjścia:

Bardziej naturalna obsługa
: Jeśli dane są w sposób naturalny podzielone na części (na przykład, są to pola wcześniej zdefiniowanego nagłówka w pliku), wówczas wektorowe operacje wejścia i wyjścia pozwalają manipulować danymi w sposób intuicyjny.

Sprawność
: Pojedyncza operacja wektorowa wejścia i wyjścia może zastąpić wiele liniowych operacji wejścia i wyjścia.

Wydajność
: Jako dodatek do zredukowanej liczby wywołań funkcji systemowych, implementacja wektorowych operacji wejścia i wyjścia może zapewnić poprawę wydajności w porównaniu z implementacją liniowych operacji wejścia i wyjścia. Poprawę tę uzyskuje się dzięki użyciu wewnętrznych optymalizacji.

Niepodzielność
: W przeciwieństwie do liniowych operacji wejścia i wyjścia dany proces może przeprowadzić pojedynczą operację wektorową wejścia i wyjścia, bez ponoszenia ryzyka przerwania jej przez inny proces.

Zarówno bardziej naturalna obsługa, jak również niepodzielność osiągalne są bez użycia mechanizmów wektorowych operacji wejścia i wyjścia. Przed zapisem proces może połączyć osobne bufory w jeden obszar pamięci, a po odczycie podzielić zwrócony bufor na mniejsze wektory — oznacza to, że aplikacja w przestrzeni użytkownika może przeprowadzać operacje rozpraszania i łączenia danych. Takie rozwiązanie nie jest jednak ani skuteczne, ani też łatwe do zaimplementowania.

Funkcje readv() i writev()

W standardzie POSIX 1003.1-2001 zostały zdefiniowane, a następnie zaimplementowane w Linuksie dwie funkcje systemowe wykonujące rozproszone operacje wejścia i wyjścia. Implementacja tych funkcji w Linuksie posiada wszystkie zalety przedstawione w poprzednim podrozdziale.

Funkcja `readv()` odczytuje określoną liczbę segmentów danych z deskryptora pliku `fd` do buforów wskazywanych przez parametr `iov`. Liczba segmentów danych przekazana zostaje w parametrze `count`:

```
#include <sys/uio.h>

ssize_t readv (int fd, const struct iovec *iov, int count);
```

Funkcja `writev()` zapisuje maksymalnie `count` segmentów danych z buforów, wskazywanych przez parametr `iov`, do deskryptora pliku `fd`:

```
#include <sys/uio.h>

ssize_t writev (int fd, const struct iovec *iov, int count);
```

Funkcje `readv()` i `writev()` działają w taki sam sposób jak odpowiednie funkcje `read()` i `write()`, ale wykonują zapis i odczyt dla wielu buforów z danymi.

Każda struktura `iovec` opisuje niezależny i oddzielny bufor, zwany *segmentem*:

```
#include <sys/uio.h>

struct iovec
{
   void *iov_base; /* wskaźnik na początek bufora */
   size_t iov_len; /* rozmiar bufora w bajtach */
};
```

Grupa segmentów zwana jest *wektorem*. Każdy segment w wektorze opisuje adres i długość bufora w pamięci, do którego powinny zostać wczytane lub z którego powinny być odczytane dane. Funkcja `readv()` przechodzi do obsługi kolejnego bufora dopiero wtedy, gdy całkowicie zostanie wypełniony poprzedni bufor o długości `iov_len`. Podobnie funkcja `writev()` musi najpierw zapisać wszystkie dane z bufora o długości `iov_len`, aby przejść do obsługi następnego. Obie funkcje obsługują segmenty w określonym porządku, rozpoczynając od `iov[0]`, następnie `iov[1]` i tak dalej, aż do `iov[count-1]`.

Wartości powrotne

W przypadku sukcesu, funkcje `readv()` oraz `writev()` zwracają odpowiednio liczbę odczytanych lub zapisanych bajtów. Liczba ta powinna być sumą wartości `iov_len` pochodzących ze wszystkich segmentów. W przypadku błędu, zwracają `-1` oraz odpowiednio ustawiają zmienną `errno`. Mogą one wykonać się z tymi samymi błędami jak funkcje systemowe `read()` i `write()`, a także odpowiednio ustawić identyczne wartości w zmiennej `errno`. Standardy definiują również dwa dodatkowe kody błędów.

Pierwszy z nich dotyczy sytuacji, gdy suma wszystkich wartości `iov_len` jest większa od `SSIZE_MAX`, a zwracany typ danych równy jest `ssize_t`. W tym przypadku nie nastąpi przesłanie danych, funkcje zwrócą `-1`, a zmienna `errno` zostanie ustawiona na `EINVAL`.

Drugi kod błędu pojawia się, gdy wartość licznika `count` jest większa od `IOV_MAX`. POSIX nakłada ograniczenia na wartość `count`: powinna być ona większa od zera, a mniejsza lub równa wartości `IOV_MAX`, która zdefiniowana jest w pliku nagłówkowym `<limits.h>`. Dla Linuksa wartość ta wynosi obecnie `1024`. Jeśli `count` równe jest zeru, funkcje systemowe również zwrócą zero[1].

[1] Należy zwrócić uwagę na to, że inne systemy uniksowe mogą ustawiać zmienną `errno` na wartość `EINVAL`, gdy parametr `count` równy jest zeru. Jest to dozwolone przez standardy, które mówią, że w przypadku, gdy wartość `count` równa jest zeru, zmienna `errno` może zostać ustawiona na `EINVAL` *lub* system może obsłużyć ten przypadek w inny (bezbłędny) sposób.

Optymalizacja wartości licznika

Podczas trwania wektorowej operacji wejścia i wyjścia jądro Linuksa musi zarezerwować pamięć na stworzenie wewnętrznych struktur danych, służących do przechowywania wszystkich segmentów. Zwykle ta rezerwacja wykonywana jest w sposób dynamiczny, w oparciu o wartość licznika `count`. Jądro dokonuje jednak pewnej optymalizacji, tworząc niewielką tablicę segmentów na stosie, która zostaje użyta, gdy wartość `count` jest wystarczająco mała, dzięki czemu nie ma potrzeby dynamicznego przydzielania pamięci dla segmentów, co w rezultacie powoduje lekką poprawę wydajności. Próg wartości `count` wynosi obecnie `8`, więc jeśli licznik jest równy lub mniejszy od `8`, wektorowe operacje wejścia i wyjścia zostaną wykonane przy użyciu metody efektywnej pamięciowo, wykorzystującej stos procesu.

Gdy wartość licznika `count` jest większa od `IOV_MAX`, nie następuje przesłanie danych, funkcje zwracają `−1`, a zmienna `errno` zostaje ustawiona na `EINVAL`.

Przykład użycia funkcji writev()

Poniżej zostanie przedstawiony prosty przykład zapisujący wektor danych, składający się z trzech elementów, z których każdy zawiera łańcuch znaków o różnej długości. Ten samodzielny program jest wystarczający, aby zaprezentować działanie funkcji `writev()`, a jednocześnie na tyle prosty, że może być przydatnym fragmentem kodu:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/uio.h>

int main ( )
{
   struct iovec iov[3];
   ssize_t nr;
   int fd, i;

   char *buf[] =
   {
      "Określenie korsarz pochodzi od słów guerre de course.\n",
      "Guerre de course oznacza w języku francuskim wojnę na szlakach handlowych.\n",
      "Korsarz jest inną nazwą pirata, używaną w Indiach Zachodnich (na Karaibach).\n"
   };

   fd = open ("buccaneer.txt", O_WRONLY | O_CREAT | O_TRUNC);
   if (fd == -1)
   {
      perror ("open");
      return 1;
   }

   /* wypełnij trzy struktury iovec */
   for (i = 0; i < 3; i++)
   {
      iov[i].iov_base = buf[i];
      iov[i].iov_len = strlen (buf[i]);
 }
```

```
    /* zapisz je wszystkie przy użyciu jednego wywołania funkcji */
    nr = writev (fd, iov, 3);
    if (nr == -1)
    {
       perror ("writev");
       return 1;
    }
    printf ("Zapisano %d bajtów\n", nr);

    if (close (fd))
    {
       perror ("close");
       return 1;
    }

    return 0;
}
```

Uruchomienie tego programu spowoduje wyprowadzenie oczekiwanego wyniku:

```
$ ./writev
Zapisano 206 bajtów
```

Wyświetlenie zawartości utworzonego pliku również się powiedzie:

```
$ cat buccaneer.txt
Określenie korsarz pochodzi od słów guerre de course.
Guerre de course oznacza w języku francuskim wojnę na szlakach handlowych.
Korsarz jest inną nazwą pirata, używaną w Indiach Zachodnich (na Karaibach).
```

Przykład użycia funkcji readv()

W tym podrozdziale przedstawiony zostanie prosty program używający funkcji systemowej `readv()`, aby odczytać wcześniej utworzony plik przy użyciu wektorowych operacji wejścia i wyjścia. Ten samodzielny przykład jest nieskomplikowanym, a zarazem kompletnym programem:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>

int main ( )
{
   char foo[48], bar[51], baz[49];
   struct iovec iov[3];
   ssize_t nr;
   int fd, i;

   fd = open ("buccaneer.txt", O_RDONLY);
   if (fd == -1)
   {
      perror ("open");
      return 1;
   }

   /* inicjalizacja struktur iovec */
   iov[0].iov_base = foo;
   iov[0].iov_len = sizeof (foo);
   iov[1].iov_base = bar;
   iov[1].iov_len = sizeof (bar);
   iov[2].iov_base = baz;
   iov[2].iov_len = sizeof (baz);
```

```
   /* czytanie danych do struktur przy użyciu jednej funkcji */
   nr = readv (fd, iov, 3);
   if (nr == -1)
   {
      perror ("readv");
      return 1;
   }

for (i = 0; i < 3; i++)
   printf ("%d: %s", i, (char *) iov[i].iov_base);

if (close (fd))
{
   perror ("close");
   return 1;
}

return 0;
}
```

Uruchomienie powyższego programu (po wcześniejszym uruchomieniu poprzedniego) spowoduje wyprowadzenie następującego wyniku:

```
$ ./readv
0: Określenie korsarz pochodzi od słów guerre de course.
1: Guerre de course oznacza w języku francuskim wojnę na szlakach handlowych.
2: Korsarz jest inną nazwą pirata, używaną w Indiach Zachodnich (na Karaibach).
```

Implementacja

Najprostsza implementacja funkcji `readv()` oraz `writev()` mogłaby zostać wykonana w przestrzeni użytkownika przy użyciu zwykłej pętli, jako kod podobny do poniższego:

```
#include <unistd.h>
#include <sys/uio.h>

ssize_t naive_writev (int fd, const struct iovec *iov, int count)
{
   ssize_t ret = 0;
   int i;

   for (i = 0; i < count; i++)
   {
      ssize_t nr;

      nr = write (fd, iov[i].iov_base, iov[i].iov_len);
      if (nr == -1)
      {
         ret = -1;
         break;
      }
      ret += nr;
   }

return ret;
}
```

Na szczęście nie jest to implementacja linuksowa: Linux implementuje `readv()` oraz `writev()` jako funkcje systemowe i wewnętrznie wykonuje rozproszone operacje wejścia i wyjścia. W rzeczywistości wszystkie operacje wejścia i wyjścia w jądrze Linuksa są wektorowe; funkcje `read()` i `write()` zaimplementowane zostały jako wektorowe operacje wejścia i wyjścia posiadające tylko jeden segment.

Interfejs odpytywania zdarzeń

Z powodu ograniczeń w działaniu funkcji `poll()` i `select()`, w wersji jądra 2.6[2] dla Linuksa wprowadzono możliwość *odpytywania zdarzeń* (ang. *event poll*, w skrócie *epoll*). Pomimo że mechanizm ten jest bardziej złożony od poprzednio omówionych dwóch interfejsów, pozwala na rozwiązanie podstawowego problemu wydajnościowego, który wcześniej wspomniane funkcje posiadają oraz udostępnia dodatkowo nowe możliwości.

Obie funkcje `poll()` oraz `select()` (omówione w rozdziale 2.) wymagają pełnej listy deskryptorów plików, by monitorować każde ich użycie. Lista musi zostać przetworzona przez jądro, aby możliwe było obserwowanie każdego deskryptora pliku. Gdy lista ta się powiększa (może ona zawierać setki, a nawet tysiące deskryptorów plików), jej przetwarzanie podczas każdego użycia deskryptora pliku staje się wąskim gardłem dla skalowalności.

Interfejs odpytywania zdarzeń zapobiega temu problemowi poprzez oddzielenie procesu rejestrowania deskryptorów plików od późniejszego ich monitorowania. Jedna funkcja systemowa inicjalizuje kontekst interfejsu odpytywania zdarzeń, druga dodaje lub usuwa z niego obserwowane deskryptory plików, a trzecia funkcja wykonuje faktyczne oczekiwanie na zdarzenie.

Tworzenie nowego egzemplarza interfejsu odpytywania zdarzeń

Kontekst interfejsu odpytywania zdarzeń tworzony jest za pomocą funkcji `epoll_create()`:

```
#include <sys/epoll.h>

int epoll_create (int size)
```

Poprawne wykonanie funkcji `epoll_create()` tworzy nowy egzemplarz interfejsu odpytywania zdarzeń oraz zwraca związany z nim deskryptor pliku. Ten deskryptor pliku nie ma żadnego powiązania z prawdziwym plikiem; jest po prostu uchwytem, który będzie używany podczas kolejnych wywołań wykorzystujących interfejs odpytywania zdarzeń. Parametr `size` jest wskazówką dla jądra, mówiącą o liczbie deskryptorów plików, które mają być monitorowane; nie jest to liczba maksymalna. Przekazanie poprawnego oszacowania spowoduje poprawę wydajności, lecz dokładna liczba nie jest wymagana. W przypadku błędu, funkcja zwraca `-1` oraz ustawia zmienną `errno` na jedną z poniższych wartości:

`EINVAL`
: Parametr `size` nie jest liczbą dodatnią.

`ENFILE`
: W systemie osiągnięto granicę maksymalnej liczby otwartych plików.

`ENOMEM`
: Brak pamięci dla zakończenia operacji.

Typowe użycie funkcji wygląda następująco:

```
int epfd;

epfd = epoll_create (100); /* zamierzamy obserwować ok. 100 deskryptorów plików */
if (epfd < 0)
   perror ("epoll_create");
```

[2] Interfejs odpytywania zdarzeń został wprowadzony w wersji rozwojowej jądra 2.5.44, a ukończony w wersji 2.5.66.

Deskryptor pliku, zwracany przez funkcję `epoll_create()`, powinien zostać usunięty za pomocą wywołania funkcji `close()` po zakończeniu operacji przepytywania zdarzeń.

Sterowanie działaniem interfejsu odpytywania zdarzeń

Funkcja systemowa `epoll_ctl()` może zostać użyta w celu dodania lub usunięcia deskryptorów plików z danego kontekstu interfejsu odpytywania zdarzeń:

```
#include <sys/epoll.h>

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

Nagłówek `<sys/epoll.h>` definiuje strukturę `epoll_event` w następujący sposób:

```
struct epoll_event
{
   __u32 events; /* zdarzenia */
   union
   {
      void *ptr;
      int fd;
      __u32 u32;
      __u64 u64;
   } data;
};
```

Poprawne uruchomienie funkcji `epoll_ctl()` umożliwia zarządzanie egzemplarzem interfejsu odpytywania zdarzeń, związanym z deskryptorem pliku `epfd`. Parametr `op` definiuje rodzaj działania, jaki ma być podjęty dla pliku określonego przez parametr `fd`. Parametr `event` dokładniej opisuje sposób działania wybranej operacji.

Oto dopuszczalne wartości parametru `op`:

`EPOLL_CTL_ADD`
: Dla egzemplarza `epoll` reprezentowanego przez parametr `epfd` oraz zdarzeń zdefiniowanych w parametrze `event`, umożliwia monitorowanie pliku związanego z deskryptorem pliku `fd`.

`EPOLL_CTL_DEL`
: Dla egzemplarza `epoll` reprezentowanego przez parametr `epfd`, usuwa monitorowanie pliku związanego z deskryptorem pliku `fd`.

`EPOLL_CTL_MOD`
: Modyfikuje istniejące monitorowanie deskryptora pliku `fd` przy użyciu uaktualnionych zdarzeń, zdefiniowanych w parametrze `event`.

Pole `events` w strukturze `epoll_event` określa, które zdarzenia powinny być monitorowane dla danego deskryptora pliku. Wiele zdarzeń może zostać połączonych przy użyciu operatora sumy bitowej. Oto ich poprawne wartości:

`EPOLLERR`
: W pliku wystąpił błąd. To zdarzenie jest zawsze monitorowane, nawet jeśli nie zostanie to jawnie określone.

EPOLLET
: Aktywuje przełączanie zboczem dla zdarzeń aktualnie monitorowanego pliku (w następnym podrozdziale Zdarzenia przełączane zboczem, a zdarzenia przełączane poziomem można zapoznać się dokładniej z tymi terminami). Domyślnie wybrane jest przełączanie poziomem.

EPOLLHUP
: Dla danego pliku wystąpiła sytuacja zawieszenia. Zdarzenie jest zawsze monitorowane, nawet jeśli nie zostanie to jawnie określone.

EPOLLIN
: Z pliku można czytać dane w trybie nieblokującym.

EPOLLONESHOT
: Po wygenerowaniu i odczytaniu zdarzenia, plik automatycznie przestanie być monitorowany. Aby ponownie włączyć monitorowanie, należy przekazać nową maskę zdarzeń, używając opcji `EPOLL_CTL_MOD`.

EPOLLOUT
: Do pliku można zapisywać dane w trybie nieblokującym.

EPOLLPRI
: Poza kolejką istnieją pilne dane, dostępne do odczytu.

Pole `data` wewnątrz struktury `event_poll` przeznaczone jest dla użytkownika, do jego prywatnych celów. Zawartość tego pola zwracana jest użytkownikowi po otrzymaniu określonego zdarzenia. Powszechną praktyką jest ustawianie pola `event.data.fd` na wartość `fd`, co umożliwia sprawdzenie w prosty sposób, który deskryptor pliku wygenerował zdarzenie.

W przypadku sukcesu, funkcja `epoll_ctl()` zwraca wartość zero. W przypadku błędu, zwraca `-1` oraz ustawia zmienną `errno` na jedną z poniższych wartości:

EBADF
: Parametr `epfd` nie określa poprawnego egzemplarza interfejsu odpytywania zdarzeń lub parametr `fd` nie jest poprawnym deskryptorem pliku.

EEXIST
: Wybrana została operacja `EPOLL_CTL_ADD`, lecz deskryptor pliku `fd` jest już związany z egzemplarzem interfejsu odpytywania zdarzeń `epfd`.

EINVAL
: Parametr `epfd` nie definiuje egzemplarza interfejsu odpytywania zdarzeń, jest równy parametrowi `fd` lub parametr `op` jest niepoprawny.

ENOENT
: Została wybrana operacja `EPOLL_CTL_MOD` lub `EPOLL_CTL_DEL`, lecz deskryptor pliku `fd` nie jest związany z egzemplarzem interfejsu odpytywania zdarzeń `epfd`.

ENOMEM
: Brak pamięci, by przeprowadzić daną operację.

EPERM
: Deskryptor pliku `fd` nie wspiera interfejsu odpytywania zdarzeń.

Poniższy kod przykładowo umożliwia monitorowanie pliku, związanego z deskryptorem pliku `fd`, w egzemplarzu interfejsu odpytywania zdarzeń `epfd`:

```
struct epoll_event event;
int ret;

event.data.fd = fd; /* później zwróć deskryptor pliku fd */
event.events = EPOLLIN | EPOLLOUT;

ret = epoll_ctl (epfd, EPOLL_CTL_ADD, fd, &event);
if (ret)
   perror ("epoll_ctl");
```

Aby zmodyfikować zdarzenie istniejące w egzemplarzu interfejsu odpytywania zdarzeń `epfd` dla pliku związanego z deskryptorem pliku `fd`, należy użyć następującego kodu:

```
struct epoll_event event;
int ret;

event.data.fd = fd; /* później zwróć deskryptor pliku fd */
event.events = EPOLLIN;

ret = epoll_ctl (epfd, EPOLL_CTL_MOD, fd, &event);
if (ret)
   perror ("epoll_ctl");
```

Odwrotnie, aby usunąć istniejące zdarzenie z egzemplarza interfejsu odpytywania zdarzeń `epfd` dla pliku związanego z deskryptorem pliku `fd`, należy użyć poniższego kodu:

```
struct epoll_event event;
int ret;

ret = epoll_ctl (epfd, EPOLL_CTL_DEL, fd, &event);
if (ret)
   perror ("epoll_ctl");
```

Należy zauważyć, że parametr `event` może mieć wartość `NULL`, gdy `op` równy jest `EPOLL_CTL_DEL`, gdyż maska zdarzeń jest wówczas nieużywana. Jądro w przypadku wersji wcześniejszych niż 2.6.9 błędnie wymusza, aby parametr ten był różny od `NULL`. W celu kompatybilności ze starszymi wersjami jądra, do funkcji należy przekazywać jakiś poprawny wskaźnik, różny od wartości `NULL`; nie zostanie on zmodyfikowany. W jądrze w wersji 2.6.9 ten błąd został poprawiony.

Oczekiwanie na zdarzenie w interfejsie odpytywania zdarzeń

Funkcja systemowa `epoll_wait()` oczekuje na wystąpienie zdarzenia dla deskryptorów plików, związanych z danym egzemplarzem interfejsu odpytywania zdarzeń:

```
#include <sys/epoll.h>

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

Po wywołaniu funkcja `epoll_wait()` oczekuje przez określony czas na pojawienie się zdarzeń dla deskryptorów plików, związanych z egzemplarzem interfejsu odpytywania zdarzeń `epfd`. Czas oczekiwania podawany jest w parametrze `timeout` i wyrażony w milisekundach. W przypadku sukcesu, parametr `events` wskazuje na obszar pamięci zawierający struktury `epoll_event`, które opisują każde zdarzenie. Maksymalna liczba zdarzeń przekazana jest w parametrze `maxevents`. Funkcja zwraca liczbę zdarzeń lub `-1` w przypadku błędu; dodatkowo dla tego przypadku zmienna `errno` zostaje ustawiona na jedną z poniższych wartości:

`EBADF`
: Parametr `epfd` oznacza błędny deskryptor pliku.

EFAULT
: Proces nie posiada uprawnienia do zapisu w obszarze pamięci wskazywanym przez parametr `events`.

EINTR
: Funkcja systemowa została przerwana przez sygnał, zanim zdążyła się wykonać.

EINVAL
: Parametr `epfd` oznacza błędny egzemplarz interfejsu odpytywania zdarzeń lub parametr `maxevents` jest mniejszy lub równy zeru.

Jeśli limit czasowy równy jest zeru, funkcja natychmiast kończy swoje działanie (nawet jeśli nie wystąpiło żadne zdarzenie) i zwraca zero. Jeśli limit czasowy wynosi `-1`, funkcja nie zakończy swojego działania, dopóki nie wystąpi jakieś zdarzenie. Po powrocie funkcji pole `events` w strukturze `epoll_event` opisuje rodzaj zdarzenia, które wystąpiło.

Pełny przykład użycia funkcji `epoll_wait()` wygląda następująco:

```
#define MAX_EVENTS 64

struct epoll_event *events;
int nr_events, i, epfd;

events = malloc (sizeof (struct epoll_event) * MAX_EVENTS);
if (!events)
{
   perror ("malloc");
   return 1;
}

nr_events = epoll_wait (epfd, events, MAX_EVENTS, -1);
if (nr_events < 0)
{
   perror ("epoll_wait");
   free (events);
   return 1;
}

for (i = 0; i < nr_events; i++)
{
   printf ("zdarzenie=%ld dla fd=%d\n", events[i].events, events[i].data.fd);
   /*
  * Obecnie dla każdego zdarzenia events[i].events można obsługiwać
  * deskryptor pliku events[i].data.fd w trybie nieblokującym.
  */
}

free (events);
```

Funkcje `malloc()` i `free()` zostaną omówione w rozdziale 8.

Zdarzenia przełączane zboczem, a zdarzenia przełączane poziomem

Jeśli pole `events` w parametrze `event`, przekazywanym do funkcji `epoll_ctl()`, zostanie ustawione na wartość `EPOLLET`, wówczas monitorowanie deskryptora pliku będzie przeprowadzane w trybie *przełączania zboczem* (ang. *edge-triggered*), w przeciwieństwie do trybu *przełączania poziomem* (ang. *level-triggered*).

Rozważmy następujące zdarzenia, występujące pomiędzy producentem a konsumentem podczas komunikacji przy użyciu potoku uniksowego:

1. Producent zapisuje 1 kB danych do potoku.
2. Konsument wykonuje funkcję systemową `epoll_wait()` dla tego potoku, oczekując, aż pojawią się w nim dane, co jednocześnie oznacza, że będą one gotowe do odczytu.

Dla monitorowania działającego w trybie przełączania poziomem wywołanie funkcji `epoll_wait()`, przeprowadzone w punkcie 2., zakończy się natychmiast, informując, że potok gotowy jest do odczytu. Dla monitorowania działającego w trybie przełączania zboczem wywołanie funkcji `epoll_wait()` nie zakończy się, dopóki nie zostanie zrealizowany punkt 1. Oznacza to, że jeśli nawet potok gotowy jest do odczytu w momencie wywołania funkcji `epoll_wait()`, będzie ona oczekiwać, dopóki dane nie zostaną zapisane do tego potoku.

Domyślnym zachowaniem jest tryb przełączania poziomem. W taki sposób działają funkcje `poll()` i `select()` — jest to zachowanie oczekiwane przez większość projektantów oprogramowania. Tryb przełączania zboczem wymaga innego podejścia do programowania, polegającego na powszechnym użyciu nieblokujących operacji wejścia i wyjścia oraz dokładnym sprawdzaniu, czy wartość `errno` równa jest `EAGAIN`.

Używana terminologia ma swoje źródło w dziedzinie elektrotechniki. Gdy na linii istnieje sygnał, pojawia się przerwanie generowane poziomem. Przerwanie generowane zboczem pojawia się tylko podczas narastającego lub opadającego zbocza, w momencie zmian w sygnale na linii. Przerwania generowane poziomem są przydatne, gdy należy odczytać stan po zdarzeniu (sygnał na linii). Przerwania generowane zboczem są przydatne, gdy ważne jest samo zdarzenie (zmiana sygnału na linii).

Odwzorowywanie plików w pamięci

Jako alternatywę dla standardowych operacji plikowych wejścia i wyjścia jądro udostępnia interfejs, który pozwala aplikacji na odwzorowanie pliku na obszar pamięci. Odwzorowywanie to zachodzi w relacji „jeden do jeden", co oznacza, że każdy adres w pamięci odpowiada elementarnej jednostce danych z pliku. Programista może więc uzyskać dostęp do pliku bezpośrednio poprzez odwołanie się do pamięci, identycznie jak jest to wykonywane w przypadku innych obszarów danych, znajdujących się w niej — jest nawet możliwe zezwolenie na taki zapis do obszaru pamięci, aby dane były przesyłane na dysk w sposób niewidoczny.

POSIX standaryzuje, a Linux implementuje funkcję systemową `mmap()`, służącą do odwzorowywania obiektów na obszar pamięci. W tym podrozdziale poddana zostanie analizie funkcja `mmap()` w kontekście przeprowadzania operacji wejścia i wyjścia na plikach odwzorowywanych w pamięci. W rozdziale 8. omówione zostaną inne zastosowania funkcji `mmap()`.

Funkcja mmap()

Wywołanie funkcji `mmap()` powoduje wysłanie do jądra żądania odwzorowania obiektu reprezentowanego przez deskryptor pliku `fd`, poczynając od położenia w pliku o wartości `offset` na obszar pamięci o wielkości przekazanej w parametrze `len`. Oba parametry `offset` i `len` posia-

dają wartości wyrażone w bajtach. Jeśli określony jest parametr `addr`, sugeruje on początkowy adres w pamięci. Uprawnienia dostępu ustalane są za pomocą parametru `prot`, a dodatkowe opcje, modyfikujące zachowanie, mogą zostać określone w parametrze `flags`:

```
#include <sys/mman.h>

void * mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset);
```

Parametr `addr` sugeruje jądru, od jakiego adresu powinno ono odwzorować plik w pamięci. Jest to tylko wskazówka; najczęściej przekazywana jest w nim wartość zero. Funkcja zwraca rzeczywisty adres w pamięci, od którego rozpoczyna się odwzorowanie.

Parametr `prot` definiuje wymagany sposób dostępu do odwzorowywanego obszaru pamięci. Może posiadać on wartość `PROT_NONE`, która oznacza, że nie istnieje dostęp dla stron pamięci z tego odwzorowania (nie ma to za bardzo sensu!), lub też składać się z jednego, lub większej liczby poniższych znaczników połączonych ze sobą za pomocą operatora sumy bitowej:

`PROT_READ`
: Strony pamięci mogą być odczytywane.

`PROT_WRITE`
: Strony pamięci mogą być zapisywane.

`PROT_EXEC`
: Strony pamięci mogą być wykonywane.

Wymagany sposób dostępu do pamięci nie może kolidować z trybem otwarcia pliku. Na przykład, jeśli program otwiera plik w trybie do odczytu, parametr `prot` nie może zawierać znacznika `PROT_WRITE`.

Znaczniki dostępu, architektury i bezpieczeństwo

Podczas gdy POSIX definiuje zestaw czterech bitów dostępu (czytanie, pisanie, wykonywanie oraz brak dostępu), niektóre architektury udostępniają jedynie jego podzbiór. Z reguły procesor nie odróżnia akcji odczytu od wykonywania. Dla tego przypadku procesor może posiadać tylko pojedynczy znacznik czytania. W takich systemach obecność znacznika `PROT_READ` wymusza działanie znacznika `PROT_EXEC`. Do tej pory jedynym systemem tego typu była architektura x86.

Oczywiście opieranie się na powyższym mechanizmie nie jest przenośne. Programy przenośne powinny zawsze ustawiać znacznik `PROT_EXEC`, jeśli zamierzają wykonywać kod odwzorowany w pamięci.

Odwrotna sytuacja jest jednym z powodów pojawiania się częstych ataków hakerskich, wykorzystujących przepełnienie bufora: nawet jeśli dane odwzorowanie nie posiada uprawnienia do wykonania, procesor może na nie zezwolić.

W nowych procesorach x86 wprowadzono bit NX (*nie wykonuj* — ang. *no-execute*), który pozwala na tworzenie odwzorowania z uprawnieniem do odczytu, ale nie do wykonania. W tych nowszych systemach obecność znacznika `PROT_READ` nie wymusza już działania znacznika `PROT_EXEC`.

Parametr `flags` opisuje rodzaj odwzorowania oraz pewne cechy jego zachowania. Może on składać się z sumy bitowej następujących znaczników:

`MAP_FIXED`
: Nakazuje funkcji `mmap()` traktowanie parametru `addr` jako konieczność, a nie wskazówkę. Jeśli jądro nie będzie zdolne do umieszczenia odwzorowania w podanym adresie, wywołanie funkcji nie powiedzie się. Jeśli parametry określające adres i długość spowodują przykrycie istniejącego odwzorowania, strony przykryte zostaną usunięte i zastąpione nowym odwzorowaniem. Ponieważ ta opcja wymaga dokładnej znajomości przestrzeni adresowej procesu, dlatego nie jest przenośna, a jej użycie nie jest zalecane.

`MAP_PRIVATE`
: Ustala, że odwzorowanie nie jest udostępnione. Plik zostaje odwzorowany przy zapisie w trybie kopiowania, a zmiany wykonywane w pamięci przez proces nie powodują odpowiednich modyfikacji w rzeczywistym pliku lub w odwzorowaniach dla innych procesów.

`MAP_SHARED`
: Udostępnia odwzorowanie wszystkim procesom, które odwzorowują ten sam plik. Zapis do odwzorowania jest równoznaczny z zapisem do pliku. Czytanie z odwzorowania uwzględni zapisy wykonane przez inne procesy.

Podany musi zostać znacznik `MAP_SHARED` albo `MAP_PRIVATE`, lecz nie oba jednocześnie. Inne, bardziej zaawansowane znaczniki zostaną omówione w rozdziale 8.

Gdy następuje odwzorowanie deskryptora pliku, licznik odwołań do pliku zostaje zwiększony. Dlatego też można zamknąć deskryptor pliku po dokonaniu odwzorowania, a proces w dalszym ciągu będzie miał do niego dostęp. Odpowiednie zmniejszenie licznika odwołań do pliku nastąpi podczas usunięcia odwzorowania lub w momencie zakończenia działania procesu.

Na przykład, poniższy fragment kodu odwzorowuje plik, reprezentowany przez deskryptor pliku `fd`, poczynając od jego pierwszego bajta. Wielkość obszaru odwzorowania określona jest w parametrze `len` i wyrażona w bajtach, a nadane uprawnienia pozwalają tylko na odczyt:

```
void *p;

p = mmap (0, len, PROT_READ, MAP_SHARED, fd, 0);
if (p == MAP_FAILED)
   perror ("mmap");
```

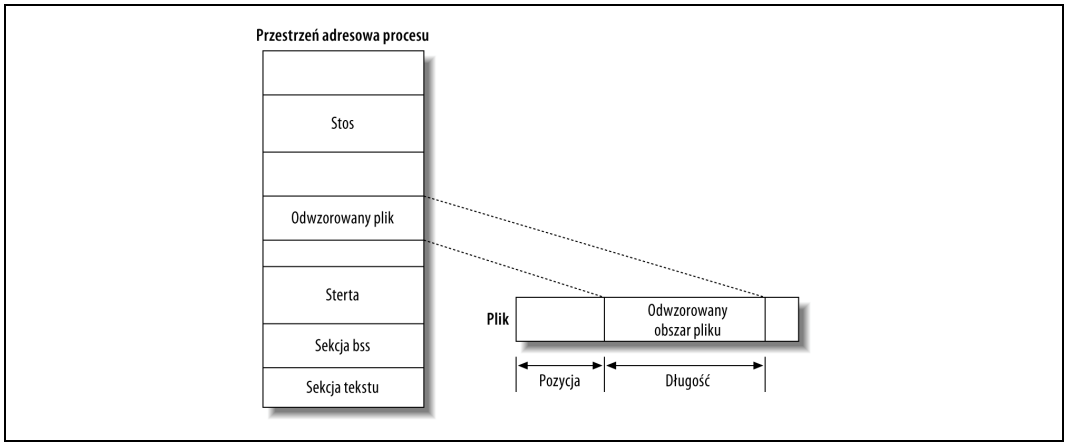
Rysunek 4.1. przedstawia wpływ parametrów funkcji `mmap()` na odwzorowanie pomiędzy plikiem, a przestrzenią adresową procesu.

Rozmiar strony

Strona jest najmniejszą jednostką pamięci, która może posiadać odrębne uprawnienia i zachowanie. Zgodnie z tym, strony są elementami tworzącymi odwzorowania w pamięci, a te z kolei są składnikami tworzącymi przestrzeń adresową procesu.

Strony używane są przez funkcję systemową `mmap()`. Oba parametry `addr` i `offset` muszą być wyrównane do granicy adresowej o wielkości strony. Oznacza to, że muszą być całkowitymi wielokrotnościami rozmiaru strony.

Zatem odwzorowania składają się z całkowitej wielokrotności stron. Jeśli parametr `len`, przekazany przez procedurę wywołującą, nie jest wyrównany do granicy strony (być może dlatego, że rozmiar odwzorowanego pliku nie jest wielokrotnością rozmiaru strony), obszar odwzorowania



Rysunek 4.1. Odwzorowanie pliku na przestrzeń adresową procesu

zostaje zaokrąglony o kolejną, pełną stronę. Bajty, znajdujące się w dodanym obszarze pamięci, pomiędzy ostatnim poprawnym adresem, a końcem obszaru odwzorowania, zostają wypełnione zerami. Jakakolwiek próba odczytu z tego obszaru zwróci zera. Każda próba zapisu do tego obszaru nie spowoduje zmian w odwzorowanym pliku, nawet jeśli rodzaj odwzorowania równy jest `MAP_SHARED`. Tylko pierwotna liczba bajtów, określona przez parametr `len`, może zostać zapisana do pliku.

Funkcja sysconf(). Standardową metodą pozwalającą na otrzymanie rozmiaru strony, zdefiniowaną w standardzie POSIX, jest funkcja `sysconf()`, która umożliwia uzyskanie różnego rodzaju informacji o systemie:

```
#include <unistd.h>

long sysconf (int name);
```

Funkcja `sysconf()` zwraca wartość odpowiedniego parametru konfiguracji lub `-1`, jeśli nazwa parametru jest niepoprawna. W przypadku błędu, funkcja ustawia zmienną `errno` na wartość `EINVAL`. Ponieważ `-1` może być poprawną wartością niektórych parametrów (na przykład ograniczeń, gdzie `-1` oznacza brak ograniczenia), rozsądnym działaniem może być zerowanie zmiennej `errno` przed wywołaniem funkcji `sysconf()`, a następnie po jej zakończeniu sprawdzanie, czy wartość się nie zmieniła.

POSIX definiuje parametr o nazwie `_SC_PAGESIZE` (oraz równoważną stałą `_SC_PAGE_SIZE`), który określa rozmiar strony w bajtach. Dlatego też pobieranie rozmiaru strony jest proste:

```
long page_size = sysconf (_SC_PAGESIZE);
```

Funkcja getpagesize(). Linux udostępnia również funkcję `getpagesize()`:

```
#include <unistd.h>

int getpagesize (void);
```

Wywołanie funkcji `getpagesize()` również zwróci rozmiar strony w bajtach. Użycie jej jest nawet prostsze niż w przypadku `sysconf()`:

```
int page_size = getpagesize ( );
```

Funkcja nie jest udostępniana we wszystkich systemach uniksowych; usunięto ją z wersji standardu POSIX 1003.1-2001. Została jednak zaprezentowana, aby zapewnić kompletność definicji.

Makro PAGE_SIZE. Rozmiar strony przechowywany jest również statycznie w postaci makra `PAGE_SIZE`, które zdefiniowane jest w pliku nagłówkowym `<asm/page.h>`. Dlatego też trzecim możliwym sposobem otrzymania rozmiaru strony jest użycie następującego kodu:

```
int page_size = PAGE_SIZE;
```

W przeciwieństwie jednak do dwóch poprzednich możliwości to podejście pozwala na uzyskanie systemowego rozmiaru strony istniejącego w momencie kompilacji, a nie w fazie wykonania programu. Niektóre architektury wspierają wiele rodzajów maszyn, posiadających różne rozmiary stron, a pewne rodzaje maszyn same nawet wspierają różne wielkości stron! Pojedynczy program binarny powinien działać we wszystkich rodzajach maszyn dla danej architektury — to znaczy, powinno się go wszędzie skompilować i uruchomić. Zakodowanie na stałe rozmiaru strony prawdopodobnie zlikwiduje tę możliwość. Dlatego też należy określać rozmiar strony w czasie wykonania programu. Ponieważ parametry `addr` i `offset` są zwykle równe zeru, wymaganie to nie jest zbyt trudne do spełnienia.

Ponadto, w przyszłych wersjach jądra makro `PAGE_SIZE` nie będzie prawdopodobnie udostępnione dla przestrzeni użytkownika. Zostało ono omówione w tym rozdziale z powodu jego częstego występowania w kodzie Uniksa, lecz programista nie powinien go używać w swoich programach. Najlepszym rozwiązaniem jest użycie funkcji `sysconf()`.

Wartości powrotne i kody błędów

W przypadku sukcesu, funkcja `mmap()` zwraca położenie odwzorowania w pamięci. W przypadku błędu, zwraca kod `MAP_FAILED` oraz odpowiednio ustawia zmienną `errno`. Wywołanie funkcji `mmap()` nigdy nie spowoduje zwrócenia wartości zero.

Możliwe wartości zmiennej `errno` są następujące:

`EACCESS`
: Podany deskryptor pliku nie oznacza pliku zwykłego lub tryb jego otwarcia koliduje z wartościami parametrów `prot` albo `flags`.

`EAGAIN`
: Plik został zablokowany przy użyciu mechanizmu blokady pliku.

`EBADF`
: Dany deskryptor pliku jest niepoprawny.

`EINVAL`
: Mamy do czynienia z jednym lub więcej parametrów `addr`, `len`, `off` nieprawidłowych.

`ENFILE`
: Osiągnięto systemowe ograniczenie liczby otwartych plików.

`ENODEV`
: System plików, w którym znajduje się odwzorowywany plik, nie zapewnia mechanizmu odwzorowania w pamięci.

`ENOMEM`
: Proces nie posiada wystarczającej pamięci do wykonania funkcji.

`EOVERFLOW`
: Wynik operacji sumy wartości parametrów `addr` i `len` przekracza rozmiar przestrzeni adresowej.

EPERM
: Wybrany został tryb dostępu PROT_EXEC, lecz system plików jest zamontowany w trybie noexec.

Dodatkowe sygnały

Z działaniami w obszarach odwzorowań związane są dwa sygnały:

SIGBUS
: Sygnał ten wygenerowany zostaje w momencie, gdy proces zamierza uzyskać dostęp do obszaru odwzorowania, który nie jest już poprawny — na przykład, gdy plik został obcięty po wykonaniu na nim operacji odwzorowania.

SIGSEGV
: Ten sygnał wygenerowany zostaje, gdy proces zamierza wykonać operację zapisu do obszaru, który odwzorowany został w trybie dostępu tylko do odczytu.

Funkcja munmap()

Linux dostarcza funkcji systemowej munmap() w celu usunięcia odwzorowania, utworzonego przez funkcję mmap():

```
#include <sys/mman.h>

int munmap (void *addr, size_t len);
```

Wywołanie funkcji munmap() usuwa dowolne odwzorowanie, które zawiera strony umieszczone w przestrzeni adresowej procesu, poczynając od adresu addr. Adres ten musi być wyrównany do rozmiaru strony. Obszar, dla którego zostaje usunięte odwzorowanie, posiada wielkość określoną w parametrze len i wyrażoną w bajtach.

Zazwyczaj funkcja munmap() używa wartości powrotnej i parametru len pochodzących z wcześniejszego wywołania funkcji mmap().

W przypadku sukcesu, funkcja munmap() zwraca wartość zero. W przypadku błędu, zwraca −1 oraz odpowiednio ustawia zmienną errno. Jedyną możliwą wartością errno jest EINVAL, która oznacza, że jeden lub więcej parametrów jest nieprawidłowych.

Poniższy fragment kodu przykładowo usuwa odwzorowania dla takich obszarów pamięci, których strony leżą w granicach [addr, addr + len]:

```
if (munmap (addr, len) == -1)
   perror ("munmap");
```

Przykład odwzorowania w pamięci

Rozważmy przykład prostego programu, który używa funkcji mmap(), aby wysłać zawartość pliku, wybranego przez użytkownika, na standardowe wyjście:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
```

```
int main (int argc, char *argv[])
{
   struct stat sb;
   off_t len;
   char *p;
   int fd;

   if (argc < 2)
   {
      fprintf (stderr, "Użycie programu: %s <plik>\n", argv[0]);
      return 1;
   }

   fd = open (argv[1], O_RDONLY);
   if (fd == -1)
   {
      perror ("open");
      return 1;
   }

   if (fstat (fd, &sb) == -1)
   {
      perror ("fstat");
      return 1;
   }

   if (!S_ISREG (sb.st_mode))
   {
      fprintf (stderr, "%s nie jest plikiem\n", argv[1]);
      return 1;
   }

   p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
   if (p == MAP_FAILED)
   {
      perror ("mmap");
      return 1;
   }

   if (close (fd) == -1)
   {
      perror ("close");
      return 1;
   }
   for (len = 0; len < sb.st_size; len++)
      putchar (p[len]);

   if (munmap (p, sb.st_size) == -1)
   {
      perror ("munmap");
      return 1;
   }

   return 0;
}
```

Jedynym, nieznanym elementem w tym przykładzie powinna być funkcja `fstat()`, która zostanie omówiona w rozdziale 7. Wszystko, co trzeba w tym momencie o niej wiedzieć, to fakt, że zwraca informacje opisujące dany plik. Makro `S_ISREG()` używa niektórych z tych informacji i pozwala się upewnić, że dany plik jest plikiem zwykłym (w przeciwieństwie do pliku urządzenia lub katalogu), zanim zostanie on odwzorowany w pamięci. Zachowanie plików innych niż zwykłe

podczas operacji odwzorowania zależy od odwzorowywanego urządzenia. Niektóre urządzenia można odwzorowywać w pamięci; pliki inne niż zwykłe nie mogą zostać odwzorowane i spowodują ustawienie zmiennej `errno` na wartość `EACCESS`.

Zalety używania funkcji mmap()

Obsługa plików przy użyciu funkcji `mmap()` posiada kilka zalet w porównaniu ze standardowymi funkcjami systemowymi `read()` i `write()`. Oto niektóre z nich:

- Czytanie lub zapisywanie do pliku odwzorowanego w pamięci chroni przed dodatkowym kopiowaniem występującym podczas zastosowania funkcji systemowych `read()` i `wrtite()`, w przypadku których dane muszą być dwukrotnie kopiowane przy użyciu bufora w przestrzeni użytkownika.
- Oprócz istnienia ewentualnego zagrożenia pojawiania się błędów strony, czytanie i pisanie do pliku odwzorowanego w pamięci nie powoduje żadnego obciążenia nadmiernym wywoływaniem funkcji systemowych lub przełączaniem kontekstu. Jest tak proste jak dostęp do pamięci.
- Gdy wiele procesów odwzorowuje ten sam obiekt w pamięci, dane są współdzielone pomiędzy nimi. Odwzorowania z uprawnieniami do odczytu oraz zapisu są współdzielone w całości; prywatne odwzorowania z uprawnieniami do zapisu udostępniają strony, które nie zostały jeszcze zapisane i działają w trybie kopiowania podczas zapisu.
- Szukanie w pliku odwzorowanym w pamięci wymaga zastosowania trywialnych operacji na wskaźnikach. Nie ma potrzeby użycia funkcji systemowej `lseek()`.

Z tych powodów wybór funkcji `mmap()` jest wskazany w przypadku wielu aplikacji.

Wady używania funkcji mmap()

Istnieje kilka uwag, o których należy pamiętać podczas używania funkcji `mmap()`:

- Odwzorowania w pamięci mają zawsze rozmiar równy wielokrotności rozmiaru strony. Dlatego też różnica pomiędzy rozmiarem odwzorowywanego pliku, a liczbą stron zostaje „zmarnowana" w postaci nieużywanego obszaru. Dla małych plików wielkość takiego obszaru może być już znaczącym fragmentem całego odwzorowania. Na przykład, dla stron o rozmiarze 4 kB odwzorowanie pliku o wielkości 7 bajtów spowoduje utratę 4089 bajtów.
- Odwzorowania w pamięci muszą zostać dopasowane do przestrzeni adresowej procesu. Dla 32-bitowej przestrzeni adresowej istnienie bardzo dużej liczby odwzorowań o różnych rozmiarach może spowodować pojawienie się fragmentacji tej przestrzeni, co z kolei zaowocuje trudnościami w odnalezieniu dużych, ciągłych i nieużywanych obszarów pamięci. Ten problem jest oczywiście mniej widoczny w 64-bitowej przestrzeni adresowej.
- Wewnątrz jądra powstaje obciążenie związane z tworzeniem i obsługą odwzorowań w pamięci oraz połączonych z nimi struktur danych. To obciążenie jest przeważnie redukowane poprzez wyeliminowanie podwójnego kopiowania, opisanego w poprzednim podrozdziale, w szczególności dla dużych i często używanych plików.

Jak widać, korzyści z użycia funkcji `mmap()` są największe, gdy rozmiar odwzorowanego pliku jest duży (wielkość obszaru nieużywanego jest niewielkim fragmentem całego odwzorowania) lub jest dokładnie podzielny przez rozmiar strony (co powoduje, że nie istnieje obszar nieużywany).

Zmiana rozmiaru odwzorowania

Linux udostępnia funkcję systemową `mremap()`, która pozwala na powiększenie lub zmniejszenie rozmiaru danego odwzorowania. Jest ona specyficzna dla Linuksa:

```
#define _GNU_SOURCE

#include <unistd.h>
#include <sys/mman.h>

void * mremap (void *addr, size_t old_size, size_t new_size, unsigned long flags);
```

Wywołanie funkcji systemowej `mremap()` powoduje, że rozmiar odwzorowania wynoszący aktualnie `[addr, addr + old_size)` zostanie powiększony lub zmniejszony do nowej wielkości, określonej w parametrze `new_size`. Jądro może w tym samym czasie ewentualnie przesunąć obszar odwzorowania w zależności od dostępności wolnej pamięci w przestrzeni adresowej procesu i wartości znaczników `flags`.

Nawias otwierający `[` w wyrażeniu `[addr, addr + old_size)` oznacza, że obszar pamięci rozpoczyna się od adresu dolnego (i go zawiera), natomiast nawias zamykający `)` wskazuje, że obszar kończy się na adresie górnym, lecz go nie zawiera. Ten sposób zapisu znany jest pod nazwą *notacji przedziałów*.

Parametr `flags` może wynosić `0` lub `MREMAP_MAYMOVE`, co oznacza, że jądro ma możliwość przesuwania obszaru odwzorowania, jeśli będzie to wymagane dla przeprowadzenia operacji zmiany rozmiaru. Zmiana większego rozmiaru powiedzie się, jeśli jądro będzie mogło przesunąć obszar odwzorowania.

Wartości powrotne i kody błędów

W przypadku sukcesu, funkcja `mremap()` zwraca wskaźnik do zmienionego obszaru odwzorowania. W przypadku błędu, zwraca `MAP_FAILED` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EAGAIN`
: Obszar pamięci jest zablokowany i jego rozmiar nie może zostać zmieniony.

`EFAULT`
: Niektóre strony dla podanego zakresu nie są poprawnymi stronami w przestrzeni adresowej procesu lub zaistniał problem podczas zmiany odwzorowania dla danych stron.

`EINVAL`
: Błędny argument.

`ENOMEM`
: Dany zakres nie może zostać poszerzony bez wykonania operacji przesunięcia (a znacznik `MREMAP_MAYMOVE` nie został podany) lub nie istnieje wolne miejsce w przestrzeni adresowej procesu.

Funkcja `mremap()` jest często używana w bibliotekach, na przykład w *glibc*, aby zaimplementować sprawną obsługę funkcji `realloc()`, która jest interfejsem używanym w celu zmiany rozmiaru bloku pamięci, otrzymanego pierwotnie za pomocą funkcji `malloc()`. Na przykład:

```
void * realloc (void *addr, size_t len)
{
   size_t old_size = look_up_mapping_size (addr);
   void *p;

   p = mremap (addr, old_size, len, MREMAP_MAYMOVE);
   if (p == MAP_FAILED)
      return NULL;

   return p;
}
```

Kod ten działałaby jedynie wtedy, gdyby wszystkie alokacje, wykonane przez `malloc()`, były unikalnymi odwzorowaniami anonimowymi; mimo to jest on użytecznym przykładem osiągnięcia poprawy wydajności. W powyższym przykładzie założono, że programista napisał funkcję `look_up_mapping_size()`.

Biblioteka GNU języka C rzeczywiście używa funkcji `mmap()` oraz do niej podobnych, aby wykonywać pewne przydziały pamięci. Proces ten zostanie dokładniej opisany w rozdziale 8.

Zmiana uprawnień odwzorowania

POSIX definiuje interfejs `mprotect()`, który pozwala programom zmieniać uprawnienia dla istniejących obszarów pamięci:

```
#include <sys/mman.h>

int mprotect (const void *addr, size_t len, int prot);
```

Wywołanie funkcji `mprotect()` spowoduje zmianę trybu uprawnień dla stron pamięci, znajdujących się w przedziale adresowym `[addr, addr + len)`, gdzie `addr` jest adresem wyrównanym do wielkości strony. Parametr `prot` może przyjmować takie same wartości jak identyczny parametr w funkcji `mmap()`: `PROT_NONE`, `PROT_READ`, `PROT_WRITE` oraz `PROT_EXEC`. Wartości te nie sumują się; jeśli obszar pamięci ma uprawnienia do czytania, a parametr `prot` zostanie ustawiony na wartość `PROT_WRITE`, wywołanie funkcji uczyni ten obszar dostępnym tylko do zapisu.

W niektórych systemach funkcja `mprotect()` może działać tylko z odwzorowaniami w pamięci, utworzonymi poprzednio przez funkcję `mmap()`. W Linuksie funkcja `mprotect()` działa z każdym obszarem pamięci.

Wartości powrotne i kody błędów

W przypadku sukcesu, funkcja `mprotect()` zwraca `0`. W przypadku błędu, zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Obszar pamięci nie może otrzymać uprawnień przekazanych w parametrze `prot`. Może się to zdarzyć wtedy, gdy podejmuje się próbę stworzenia odwzorowania z uprawnieniami do zapisu dla pliku otwartego w trybie tylko do odczytu.

`EINVAL`
: Parametr `addr` jest niepoprawny lub niewyrównany do granicy strony.

ENOMEM
: Dla jądra nie istnieje wystarczająca ilość pamięci, aby zakończyć z powodzeniem wykonanie funkcji albo jedna lub więcej stron w podanym obszarze pamięci nie jest poprawnym fragmentem przestrzeni adresowej procesu.

Synchronizacja odwzorowanego pliku

POSIX udostępnia równoważnik funkcji systemowej `fsync()`, omówionej w rozdziale 2., działający z plikami odwzorowanymi w pamięci:

```
#include <sys/mman.h>

int msync (void *addr, size_t len, int flags);
```

Wywołanie funkcji `msync()` spowoduje zapisanie na dysk wszystkich zmian wykonanych w pliku odwzorowanym za pomocą funkcji `mmap()`, przez co zostanie uzyskana synchronizacja pliku z jego odwzorowaniem. Na dysku zostaje zapisany zsynchronizowany plik lub fragment pliku związany z odwzorowaniem rozpoczynającym się od adresu `addr` i posiadającym długość przekazaną w parametrze `len`, wyrażoną w bajtach. Parametr `addr` musi być wyrównany do granicy strony; najczęściej jest to wartość powrotna z wcześniejszego wywołania funkcji `mmap()`.

Bez wywołania funkcji `msync()` nie można zagwarantować, że zmodyfikowane obszary pamięci zostaną zapisane na dysk, zanim odwzorowanie przestanie istnieć. Różni się to od zachowania w przypadku funkcji `write()`, w którym bufor zaznaczany jest jako zmodyfikowany podczas operacji zapisu oraz umieszczany w kolejce, oczekując na zrzucenie go na dysk. Podczas zapisu do odwzorowania w pamięci proces modyfikuje bezpośrednio strony pliku w buforze stron jądra bez udziału samego jądra. Synchronizowanie bufora stron z dyskiem, wykonywane przez jądro, może nie nastąpić zbyt szybko.

Parametr `flag` modyfikuje sposób działania operacji synchronizowania danych. Jest to suma bitowa następujących wartości:

`MS_ASYNC`
: Określa, że synchronizacja powinna zostać wykonana asynchronicznie. Następuje zaplanowanie aktualizacji danych, lecz funkcja `msync()` natychmiast kończy swoje działanie bez czekania na sam proces zapisu.

`MS_INVALIDATE`
: Oznacza, że wszystkie inne buforowane kopie danych dla danego odwzorowania muszą zostać unieważnione. Każdy nowy dostęp do dowolnego odwzorowania w danym pliku spowoduje użycie zawartości zsynchronizowanej ostatnio z dyskiem.

`MS_SYNC`
: Określa, że operacja powinna zostać wykonana synchronicznie. Funkcja `msync()` nie zakończy swojego działania, dopóki wszystkie strony nie zostaną zapisane na dysk.

Przy wywołaniu funkcji powinna zostać użyta jedna z opcji `MS_ASYNC` lub `MS_SYNC`, lecz nie obie jednocześnie.

Użycie funkcji jest proste:

```
if (msync (addr, len, MS_ASYNC) == -1)
   perror ("msync");
```

Powyższy przykład synchronizuje w trybie asynchronicznym obszar na dysku z plikiem odwzorowanym w przestrzeni adresowej procesu o zakresie [addr, addr + len).

Wartości powrotne i kody błędów

W przypadku sukcesu, funkcja msync() zwraca 0. W przypadku błędu, zwraca −1 oraz odpowiednio ustawia zmienną errno na jedną z poniższych wartości:

EINVAL
: Parametr flag posiada ustawione oba znaczniki MS_ASYNC i MS_SYNC, został ustawiony niepoprawny znacznik lub parametr addr nie jest wyrównany do granicy strony.

ENOMEM
: Podany obszar pamięci (lub jego fragment) nie jest odwzorowany. Należy zauważyć, że Linux zwróci w tym przypadku ENOMEM, zgodnie z wymaganiem standardu POSIX, lecz mimo tego zsynchronizuje każde inne poprawne odwzorowanie dla danego obszaru pamięci.

Przed pojawieniem się wersji jądra Linuksa 2.4.19, funkcja msync() zwracała EFAULT zamiast ENOMEM.

Dostarczanie porad dotyczących odwzorowania w pamięci

Linux udostępnia funkcję systemową madvise(), która umożliwia procesom przekazywanie porad i wskazówek do jądra, aby poinformować go, w jaki sposób odwzorowanie będzie używane. Jądro może zoptymalizować swoje działanie wykorzystując informacje dotyczące zamierzonego sposobu używania odwzorowania. Jądro Linuksa dynamicznie modyfikuje swój sposób działania oraz pozwala na uzyskanie optymalnej wydajności bez potrzeby jawnego przekazywania porad, dlatego też ich dostarczanie może przyczynić się do tego, że zamierzony sposób buforowania oraz odczyty z wyprzedzeniem na pewno będą wykonywać się poprawnie podczas różnych warunków obciążenia.

Wywołanie funkcji madvise() przekazuje poradę do jądra, informując je, w jaki sposób powinno obsługiwać obszar stron w pamięci rozpoczynający się od adresu addr i posiadający wielkość określoną w parametrze len, wyrażoną w bajtach:

```
#include <sys/mman.h>

int madvise (void *addr, size_t len, int advice);
```

Jeśli parametr len wynosi 0, jądro zastosuje poradę dla całego odwzorowania rozpoczynającego się od adresu addr. Parametr advice opisuje poradę, która może być równa jednej z poniższych wartości:

MADV_NORMAL
: Aplikacja nie posiada określonej porady do przekazania dla podanego obszaru pamięci. Odwzorowanie powinno być obsługiwane w zwykły sposób.

MADV_RANDOM
: Aplikacja zamierza korzystać ze stron w podanym obszarze pamięci w sposób swobodny (niesekwencyjny).

MADV_SEQUENTIAL
: Aplikacja zamierza korzystać ze stron w podanym obszarze pamięci w sposób sekwencyjny, w kolejności od adresów niższych do wyższych.

`MADV_WILLNEED`
: Aplikacja zamierza wkrótce skorzystać ze stron w podanym obszarze pamięci.

`MADV_DONTNEED`
: Aplikacja nie zamierza wkrótce skorzystać ze stron w podanym obszarze pamięci.

Zachowanie jądra, które pojawi się w odpowiedzi na przekazane porady, zależy od implementacji: POSIX wymaga jedynie, aby istniała funkcja przekazująca porady, natomiast nie wskazuje na ewentualne konsekwencje ich przekazania. Aktualna wersja jądra 2.6 może zachować się w następujący sposób po otrzymaniu porad przekazanych w parametrze `advise`:

`MADV_NORMAL`
: Jądro zachowuje się jak dotychczas, wykonując ograniczoną liczbę operacji odczytów z wyprzedzeniem.

`MADV_RANDOM`
: Jądro wyłącza opcję odczytów z wyprzedzeniem, odczytując jedynie minimalną ilość danych podczas każdej operacji.

`MADV_SEQUENTIAL`
: Jądro wykonuje znaczną liczbę operacji odczytu z wyprzedzeniem.

`MADV_WILLNEED`
: Jądro rozpoczyna operację odczytu z wyprzedzeniem, wczytując pewną liczbę stron do pamięci.

`MADV_DONTNEED`
: Jądro zwalnia wszystkie zasoby związane z danymi stronami oraz usuwa wszystkie zmodyfikowane i jeszcze niezsynchronizowane strony. Następna próba dostępu do odwzorowanych danych spowoduje, że będą one ponownie pobrane z pliku na dysku i umieszczone na stronach pamięci.

Typowe użycie funkcji `madvice()` jest następujące:

```
int ret;

ret = madvise (addr, len, MADV_SEQUENTIAL);
if (ret < 0)
   perror ("madvise");
```

Ten kod informuje jądro, że proces zamierza korzystać z obszaru pamięci `[addr, addr + len)` w sposób sekwencyjny.

Wartości powrotne i kody błędów

W przypadku sukcesu, funkcja `madvise()` zwraca `0`. W przypadku błędu, zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EAGAIN`
: Wewnętrzny zasób jądra (prawdopodobnie pamięć) nie był dostępny. Proces może ponowić wywołanie funkcji `madvise()`.

`EBADF`
: Obszar pamięci istnieje, lecz nie odwzorowuje pliku.

Odczyt z wyprzedzeniem

Gdy jądro Linuksa odczytuje pliki z dysku, przeprowadza wówczas optymalizację zwaną *odczytem z wyprzedzeniem*. Oznacza to, że podczas operacji odczytu danego fragmentu pliku jądro dodatkowo odczytuje również następny jego fragment. Jeśli kolejna operacja odczytu będzie musiała pobrać następny fragment — tak jak dzieje się to w przypadku, gdy istnieje sekwencyjny dostęp do pliku — jądro natychmiast zwróci żądane dane. Ponieważ dyski posiadają bufory ścieżek (zasadniczo dyski twarde przeprowadzają wewnętrznie własne operacje odczytu z wyprzedzeniem), a pliki są przeważnie sekwencyjnie umieszczane na dysku, dlatego też taka optymalizacja niewiele kosztuje.

Niektóre odczyty z wyprzedzeniem są naprawdę korzystne, lecz uzyskanie optymalnych rezultatów zależy od zakresu przeprowadzanego odczytu z wyprzedzeniem. Plik z dostępem sekwencyjnym może skorzystać z większego okna odczytu z wyprzedzeniem, podczas gdy dla pliku z dostępem swobodnym odczyt z wyprzedzeniem może wprowadzić dodatkowe obciążenie.

Analiza przeprowadzona w podrozdziale Organizacja wewnętrzna jądra, znajdującym się w rozdziale 2., wykazała, że jądro dynamicznie dopasowuje rozmiar okna dla odczytu z wyprzedzeniem w odpowiedzi na liczbę prób dostępu dla tego okna. Więcej prób dostępu sugeruje, że powiększenie okna może być korzystne; mniejsza liczba prób zaleca zmniejszenie wielkości okna. Funkcja systemowa `madvise()` pozwala aplikacjom bezpośrednio oddziaływać na rozmiar okna.

`EINVAL`
: Parametr `len` jest ujemny, `addr` nie jest wyrównany do strony, `advice` jest błędny lub strony zablokowano albo udostępniono przy użyciu znacznika `MADV_DONTNEED`.

`EIO`
: Wystąpił wewnętrzny błąd wejścia i wyjścia podczas użycia znacznika `MADV_WILLNEED`.

`ENOMEM`

Dany obszar pamięci nie jest poprawnym odwzorowaniem w przestrzeni adresowej procesu lub użyto znacznika `MADV_WILLNEED`, lecz nie istnieje wystarczająca ilość pamięci, żeby utworzyć strony w podanym obszarze.

Porady dla standardowych operacji plikowych wejścia i wyjścia

W poprzednim podrozdziale omówiono dostarczanie porad dotyczących odwzorowania w pamięci. W tym podrozdziale analizie poddane zostanie dostarczanie porad do jądra, dotyczących operacji plikowych wejścia i wyjścia. W tym celu Linux udostępnia dwa interfejsy — są to funkcje systemowe `posix_fadvise()` i `readahead()`.

Funkcja systemowa posix_fadvise()

Interfejs, udostępniający pierwszą poradę (jak wskazuje na to jego nazwa[3]), znormalizowany został przez POSIX 1003.1-2003:

```
#include <fcntl.h>

int posix_fadvise (int fd, off_t offset, off_t len, int advice);
```

Wywołanie funkcji `posix_fadvise()` dostarczy jądru wskazówki `advice` dotyczącej deskryptora pliku `fd` i położenia w pliku o zakresie `[offset, offset + len)`. Jeśli parametr `len` jest równy zeru, wskazówka zostanie zastosowana dla zakresu `[offset, długość pliku]`. Powszechnym zwyczajem jest podawanie zera w parametrach `len` i `offset`, co powoduje, że wskazówka jest stosowana dla całego pliku.

Możliwe opcje parametru `advice` są podobne do tych, które zostały użyte w przypadku funkcji `madvise()`. Parametr `advice` może być równy dokładnie jednej z poniżej podanych wartości:

`POSIX_FADV_NORMAL`
: Aplikacja nie posiada określonej porady do przekazania dla podanego obszaru pliku. Odwzorowanie powinno być obsługiwane w zwykły sposób.

`POSIX_FADV_RANDOM`
: Aplikacja zamierza korzystać z danych w podanym obszarze pliku w sposób swobodny (niesekwencyjny).

`POSIX_FADV_SEQUENTIAL`
: Aplikacja zamierza korzystać z danych w podanym obszarze pliku w sposób sekwencyjny, w kolejności od adresów niższych do wyższych.

`POSIX_FADV_WILLNEED`
: Aplikacja zamierza wkrótce skorzystać z danych w podanym obszarze pliku.

`POSIX_FADV_NOREUSE`
: Aplikacja zamierza wkrótce jednorazowo skorzystać z danych w podanym obszarze pliku.

`POSIX_FADV_DONTNEED`
: Aplikacja nie zamierza w najbliższym czasie skorzystać z danych w podanym obszarze pliku.

Podobnie jak ma to miejsce w przypadku funkcji `madvice()`, faktyczna odpowiedź na podaną wskazówkę zależy od implementacji — nawet różne wersje jądra Linuksa mogą zachowywać się odmiennie. Poniżej podano opis aktualnych odpowiedzi na podawanie wskazówek:

`POSIX_FADV_NORMAL`
: Jądro zachowuje się jak dotychczas, wykonując ograniczoną liczbę operacji odczytów z wyprzedzeniem.

`POSIX_FADV_RANDOM`
: Jądro wyłącza opcję odczytów z wyprzedzeniem, odczytując jedynie minimalną ilość danych podczas każdej operacji fizycznego czytania.

[3] `fadvise` — skrót od angielskiego *first advise* (*pierwsza porada*) — *przyp. tłum.*

POSIX_FADV_SEQUENTIAL
: Jądro wykonuje intensywne odczyty z wyprzedzeniem, podwajając rozmiar okna dla tych operacji.

POSIX_FADV_WILLNEED
: Jądro rozpoczyna operację odczytu z wyprzedzeniem, odczytując pewną liczbę stron do pamięci.

POSIX_FADV_NOREUSE
: Zachowanie jest identyczne jak dla znacznika POSIX_FADV_WILLNEED; następne wersje jądra mogą przeprowadzać dodatkową optymalizację, by dopasować się do przypadku „jednorazowego użycia". Ta wskazówka nie ma swojego odpowiednika w funkcji madvise().

POSIX_FADV_DONTNEED
: Jądro usuwa dla podanego obszaru wszystkie buforowane dane z bufora stron. Należy zauważyć, że ta wskazówka, w przeciwieństwie do innych, wymusza inne zachowanie w porównaniu z jej odpowiednikiem w funkcji madvise().

Jako przykład użycia funkcji podano poniższy fragment kodu, który informuje jądro, że cały plik reprezentowany przez deskryptor pliku fd, będzie używany w trybie swobodnym (niesekwencyjnym):

```
int ret;

ret = posix_fadvise (fd, 0, 0, POSIX_FADV_RANDOM);
if (ret == -1)
   perror ("posix_fadvise");
```

Wartości powrotne i kody błędów

W przypadku sukcesu, funkcja posix_fadvice() zwraca 0. W przypadku błędu, zwraca −1 oraz ustawia zmienną errno na jedną z poniższych wartości:

EBADF
: Podany deskryptor pliku jest niepoprawny.

EINVAL
: Podana wskazówka jest niepoprawna, deskryptor pliku dotyczy potoku lub wskazówka nie może być zastosowana dla danego pliku.

Funkcja systemowa readahead()

Funkcja systemowa posix_fadvice() jest nową funkcją dla wersji 2.6 jądra Linuksa. Poprzednio używana była funkcja readahead(), która dostarczała zachowania identycznego z użyciem funkcji posix_fadvice() ze wskazówką POSIX_FADV_WILLNEED. W przeciwieństwie do posix_
↪fadvice() funkcja readahead() jest specyficzna dla Linuksa:

```
#include <fcntl.h>

ssize_t readahead (int fd, off64_t offset, size_t count);
```

Wywołanie funkcji readahead() wypełnia bufor stron o wielkości obszaru [offset, offset + count) danymi z pliku, reprezentowanego przez deskryptor pliku fd.

Wartości powrotne i kody błędów

W przypadku sukcesu, funkcja `readahead()` zwraca zero. W przypadku błędu, zwraca `-1` oraz ustawia zmienną `errno` na jedną z poniższych wartości:

`EBADF`
: Podany deskryptor pliku jest niepoprawny.

`EINVAL`
: Podany deskryptor pliku nie odwzorowuje pliku wspierającego czytanie z wyprzedzeniem.

Porada jest tania

Zarządzanie obciążeniem może zostać znacząco poprawione dla niektórych aplikacji poprzez przekazanie niewielkiej, choć cennej porady do jądra. Dzięki takiej wskazówce można z sukcesem zmniejszać obciążenie powodowane przez operacje wejścia i wyjścia. Ponieważ dyski twarde są powolne, a nowoczesne procesory szybkie, więc nawet najmniejszy element przyczynia się do częściowego zniwelowania tej różnicy i dlatego też właściwa wskazówka może wiele pomóc.

Przed odczytaniem fragmentu pliku proces może wysłać wskazówkę `POSIX_FADV_WILLNEED`, aby nakazać jądru wczytanie pliku do bufora stron. Operacje wejścia i wyjścia zostaną przeprowadzone asynchronicznie, w tle. Gdy aplikacja ostatecznie uzyska dostęp do pliku, procedura będzie mogła zakończyć się bez generowania blokujących operacji wejścia i wyjścia.

I odwrotnie, po odczycie lub zapisie dużej ilości danych — na przykład podczas strumieniowego przesyłania obrazu wideo na dysk — proces może przekazać wskazówkę `POSIX_FADV_DONTNEED`, aby nakazać jądru usunięcie danego fragmentu pliku z bufora stron. Poważne operacje przesyłania strumieniowego mogą w sposób ciągły zapełniać bufor stron. Jeśli aplikacja nie będzie już ponownie próbować uzyskać dostępu do zapisanych danych, oznacza to, że bufor stron zostanie zapełniony zbędnymi informacjami, kosztem utraty miejsca na przechowywanie bardziej użytecznych danych. Dlatego też ma sens działanie, aby aplikacja strumieniowego przesyłania obrazu wideo okresowo wymuszała wyrzucanie danych z bufora.

Proces, który zamierza czytać cały plik, powinien dostarczyć wskazówki `POSIX_FADV_SEQUENITAL` nakazującej jądru wykonywanie szczególnie intensywnych odczytów z wyprzedzeniem. I odwrotnie, proces, który zamierza poruszać się po pliku w sposób swobodny, używając operacji szukania w różnych jego obszarach, może dostarczyć wskazówki `POSIX_FADV_RANDOM` informującej jądro, że odczyt z wyprzedzeniem będzie w tym przypadku jedynie bezwartościowym obciążeniem systemu.

Operacje zsynchronizowane, synchroniczne i asynchroniczne

W systemach uniksowych używa się często pojęć „zsynchronizowany", „niezsynchronizowany", „synchroniczny", „asynchroniczny" bez zwrócenia jednocześnie uwagi na to, że są one dość mylące — w języku polskim różnica pomiędzy słowami „synchroniczny", a „zsynchronizowany" nie jest zbyt duża!

Synchroniczna operacja zapisu nie zakończy się, dopóki zapisywane dane nie zostaną umieszczone *przynajmniej* w podręcznym buforze jądra. Synchroniczna operacja odczytu nie zakończy się, dopóki odczytywane dane nie zostaną umieszczone w buforze z przestrzeni użytkownika, dostarczonym przez aplikację. Z drugiej strony, *asynchroniczna* operacja zapisu może się zakończyć, zanim dane w ogóle pojawią się w przestrzeni użytkownika; asynchroniczna operacja odczytu może się zakończyć, zanim odczytywane dane staną się dostępne. Oznacza to, że operacje zostaną umieszczone w kolejce, aby wykonać się w późniejszym czasie. Oczywiście w tym przypadku musi istnieć pewien mechanizm, by określać, kiedy i z jakim poziomem sukcesu operacja została faktycznie wykonana.

Operacja *zsynchronizowana* jest bardziej restrykcyjna i bezpieczniejsza od zwykłej operacji synchronicznej. Zsynchronizowana operacja zapisu powoduje wyrzucenie danych na dysk, dzięki czemu uzyskuje się pewność, że informacje na dysku są zawsze zsynchronizowane z odpowiadającymi im danymi z buforów jądra. Zsynchronizowana operacja odczytu zawsze zwraca najbardziej aktualną kopię danych otrzymaną prawdopodobnie bezpośrednio z dysku.

Reasumując, pojęcia „synchroniczny" i „asynchroniczny" odnoszą się do tego, czy operacje wejścia i wyjścia oczekują na jakieś zdarzenie (np. zapisywanie danych), zanim się zakończą. Natomiast pojęcia „zsynchronizowany" i „niezsynchronizowany" określają dokładnie, jakie zdarzenie musi zaistnieć (np. zapisanie danych na dysk).

Zazwyczaj uniksowe operacje zapisu są synchroniczne i niezsynchronizowane; operacje odczytu są synchroniczne i zsynchronizowane[4]. W przypadku operacji zapisu możliwy jest każdy rodzaj połączenia tych charakterystyk, tak jak pokazano w tabeli 4.1.

Tabela 4.1. Rodzaje synchronizacji dla operacji zapisu

	Operacja zsynchronizowana	**Operacja niezsynchronizowana**
Operacja synchroniczna	Funkcje wykonujące operacje zapisu nie zostaną zakończone, dopóki dane nie będą zapisane na dysku. Jest to sposób działania dla przypadku, gdy podczas otwierania pliku użyty został znacznik `O_SYNC`.	Funkcje wykonujące operacje zapisu nie zostaną zakończone, dopóki dane nie będą zapisane w buforach jądra. Jest to zachowanie standardowe.
Operacja asynchroniczna	Funkcje wykonujące operacje zapisu zakończą się, gdy tylko żądanie zostanie umieszczone w kolejce. Gdy operacja zapisu zostanie ostatecznie wykonana, dane na pewno znajdować się będą na dysku.	Funkcje wykonujące operacje zapisu zakończą się, gdy tylko żądanie zostanie umieszczone w kolejce. Gdy operacja zapisu zostanie ostatecznie wykonana, dane na pewno znajdować się będą przynajmniej w buforach jądra.

Operacje odczytu są zawsze zsynchronizowane, ponieważ czytanie nieaktualnych danych nie ma sensu. Mogą być one jednak operacjami synchronicznymi lub asynchronicznymi, jak pokazano w tabeli 4.2.

W rozdziale 2. omówiono, w jaki sposób należy przeprowadzać zsynchronizowane operacje zapisu (używając znacznika `O_SYNC`) oraz jak można zapewnić, że wszystkie operacje wejścia

[4] Z technicznego punktu widzenia operacje odczytu, podobnie jak operacje zapisu, są również niezsynchronizowane, lecz jądro gwarantuje, że bufor stron zawiera aktualne dane. Oznacza to, że dane w buforze stron są zawsze takie same lub nowsze od danych na dysku. Dzięki temu *w praktyce* sposób działania operacji odczytu jest zawsze zsynchronizowany. Istnieją mało istotne argumenty za tym, że operacje te zachowują się w inny sposób.

Tabela 4.2. Rodzaje synchronizacji dla operacji odczytu

	Operacja zsynchronizowana
Operacja synchroniczna	Funkcje wykonujące operacje odczytu nie zakończą się, dopóki aktualne dane nie zostaną zapisane w udostępnionym buforze (jest to standardowy sposób działania).
Operacja asynchroniczna	Funkcje wykonujące operacje odczytu zakończą się, gdy tylko żądanie zostanie umieszczone w kolejce, lecz aktualne dane zostaną zwrócone wówczas, gdy operacja odczytu zostanie ostatecznie wykonana.

i wyjścia będą zsynchronizowane w danym momencie (poprzez użycie funkcji `fsync()` i jej podobnych). Obecnie przedstawione zostaną metody pozwalające na wykonywanie asynchronicznych operacji zapisu i odczytu.

Asynchroniczne operacje wejścia i wyjścia

Wykonywanie asynchronicznych operacji wejścia i wyjścia wymaga wsparcia jądra dla najniższych warstw. POSIX 1003.1-2003 definiuje interfejsy *aio*, które na szczęście zostały zaimplementowane w Linuksie. Biblioteka *aio* dostarcza zestawu funkcji pozwalających na wysyłanie żądań wykonania asynchronicznych operacji wejścia i wyjścia i otrzymywanie powiadomień po ich zakończeniu:

```
#include <aio.h>

/* blok kontrolny asynchronicznych operacji wejścia i wyjścia */
struct aiocb
{
   int aio_filedes; /* deskryptor pliku */
   int aio_lio_opcode; /* operacja do wykonania */
   int aio_reqprio; /* wartość żądanego priorytetu */
   volatile void *aio_buf; /* wskaźnik do bufora */
   size_t aio_nbytes; /* długość obszaru danych */
   struct sigevent aio_sigevent; /* numer i wartość sygnału */

   /* tu następują wewnętrzne elementy prywatne... */
};

int aio_read (struct aiocb *aiocbp);
int aio_write (struct aiocb *aiocbp);
int aio_error (const struct aiocb *aiocbp);
int aio_return (struct aiocb *aiocbp);
int aio_cancel (int fd, struct aiocb *aiocbp);
int aio_fsync (int op, struct aiocb *aiocbp);
int aio_suspend (const struct aiocb * const cblist[], int n,
  const struct timespec *timeout);
```

Asynchroniczne operacje wejścia i wyjścia oparte na wątkach

Linux udostępnia operacje *aio* jedynie dla plików otwartych przy użyciu znacznika `O_DIRECT`. Aby przeprowadzić asynchroniczne operacje wejścia i wyjścia dla plików zwykłych, otwartych bez użycia znacznika `O_DIRECT`, należy samemu zaimplementować rozwiązanie tego problemu. Bez wsparcia ze strony jądra można jedynie mieć nadzieję, iż rozwiązanie to jest na tyle bliskie działaniu asynchronicznych operacji wejścia i wyjścia, że osiągnięte rezultaty będą zbliżone do ideału.

Najpierw należy zastanowić się, dlaczego projektant oprogramowania chciałby używać asynchronicznych operacji wejścia i wyjścia:

- By wykonywać nieblokujące operacje wejścia i wyjścia.
- By oddzielić od siebie czynności umieszczania operacji wejścia i wyjścia w kolejce, dostarczania ich do jądra oraz otrzymywania potwierdzenia o ich zakończeniu.

Pierwszy punkt dotyczy wydajności. Jeśli operacje wejścia i wyjścia nigdy się nie blokują, obciążenie powodowane przez nie jest zerowe, a proces nie musi być z nimi związany. Drugi punkt uzależniony jest od procedury — innej metody obsługi operacji wejścia i wyjścia.

Najpowszechniejszym sposobem, by osiągnąć powyższe cele, jest użycie wątków (tematy dotyczące szeregowania omówione zostaną w rozdziałach 5. oraz 6.). Ten sposób podejścia wymaga wykonania następujących czynności programistycznych:

1. Należy stworzyć pulę „procesów roboczych" w celu obsługi wszystkich operacji wejścia i wyjścia.
2. Następnie zaimplementować taki zestaw interfejsów, aby pozwalały one na umieszczanie operacji wejścia i wyjścia w kolejce roboczej.
3. Jeden z tych interfejsów powinien zwracać deskryptor, jednoznacznie identyfikujący związaną z nim operację wejścia i wyjścia. W każdym wątku roboczym należy pobierać zlecenia operacji wejścia i wyjścia znajdujące się na początku kolejki i przekazywać je do wykonania, czekając na ich zakończenie.
4. Po zakończeniu należy umieszczać wyniki operacji (wartości powrotu, kody błędów, dowolne odczytane dane) w kolejce wyników.
5. Wreszcie należy zaimplementować zestaw interfejsów, w celu odzyskiwania informacji o stanie z kolejki wyników, przy użyciu pierwotnie zwróconych deskryptorów operacji wejścia i wyjścia, dzięki którym będzie możliwa identyfikacja każdej operacji.

Zrealizowanie powyższych punktów pozwoli osiągnąć zachowanie podobne do tego, które istnieje dla interfejsów *aio* w standardzie POSIX, aczkolwiek przy zwiększonym obciążeniu wynikającym z obsługi wątków.

Zarządcy operacji wejścia i wyjścia oraz wydajność operacji wejścia i wyjścia

W nowoczesnym systemie różnica pomiędzy wydajnością dysków, a wydajnością pozostałych elementów jest całkiem duża i ciągle się powiększa. Składnikiem, który powoduje największe straty wydajności, jest proces przemieszczania głowicy odczytująco-zapisującej z jednego obszaru dysku na inny, czyli operacja zwana *wyszukiwaniem*. W świecie, w którym czas wykonania wielu operacji wynosi kilka cykli procesora (a każdy może być równy jednej trzeciej nanosekundy), pojedyncza operacja wyszukiwania może trwać ponad osiem milisekund — oczywiście jest to nieduża wartość, lecz *25 milionów razy większa niż pojedynczy cykl procesora*!

Przy takiej różnicy w wydajności pomiędzy napędami dysków a resztą systemu byłoby zachowaniem naprawdę niewiarygodnie prymitywnym i nieefektywnym, gdyby wysyłać zlecenia operacji wejścia i wyjścia w takiej kolejności, w jakiej się one pojawiają. Dlatego też w jądrach nowoczesnych systemów operacyjnych implementuje się mechanizmy *zarządców operacji wejścia i wyjścia* (ang. *I/O schedulers*), których zadaniem jest minimalizacja liczby i wielkości wyszukiwań

na dysku poprzez modyfikowanie kolejności i czasu obsługi operacji wejścia i wyjścia. Zarządcy operacji wejścia i wyjścia pracują ciężko, by zmniejszyć obciążenie związane z dostępem do dysku.

Adresowanie dysku

Aby zrozumieć rolę zarządcy operacji wejścia i wyjścia, niezbędna jest pewna podstawowa wiedza. Dyski twarde adresują swoje dane przy użyciu znanego, geometrycznego podziału na cylindry, głowice, sektory, zwanego też *adresowaniem CHS* (ang. *CHS addressing*). Dysk twardy składa się z wielu talerzy, z których każdy zawiera pojedynczy dysk, trzpień obrotowy oraz głowicę odczytująco–zapisującą. Można wyobrazić sobie, że każdy talerz to płyta CD (lub płyta gramofonowa), natomiast zbiór talerzy w dysku to stos płyt CD. Każdy talerz podzielony jest na okrągłe pierścienie, podobnie jak w płycie CD, zwane *ścieżkami*. Każda ścieżka składa się z pewnej liczby sektorów.

Aby ustalić położenie określonego fragmentu danych na dysku, logika sterująca wymaga dostarczenia trzech informacji: numerów cylindra, głowicy oraz sektora. Numer cylindra określa ścieżkę, na której znajdują się dane. Jeśli talerze zostaną nałożone jeden na drugim, ścieżka przyjmie formę cylindra, uwzględniając przekrój całej konstrukcji. Inaczej mówiąc, w przypadku takiej samej odległości od środka każdego dysku, cylinder reprezentowany jest przez ścieżkę. Numer głowicy dokładnie określa głowicę odczytująco–zapisującą (co równe jest wskazaniu na określony talerz). Wyszukiwanie zostaje teraz ograniczone do pojedynczej ścieżki na pojedynczym talerzu. Następnie dysk używa numeru sektora, aby ustalić położenie danego sektora na ścieżce. Wyszukiwanie jest zakończone: dysk twardy zna wartości numerów talerza, ścieżki i sektora, aby poprawnie wyszukać dane. Może więc umieścić głowicę odczytująco–zapisującą na właściwym talerzu nad poprawną ścieżką oraz czytać i pisać do wymaganego sektora.

Na szczęście technologia nowoczesnych dysków twardych nie wymusza w komputerach komunikowania się z nimi przy użyciu pojęć cylindrów, głowic i sektorów. Współczesne dyski twarde przyporządkowują unikalny *numer bloku* (zwany również *blokiem fizycznym* lub *blokiem urządzenia*) dla każdego zestawu cylinder/głowica/sektor — blok odwzorowany jest w rzeczywistości na określony sektor. Nowoczesne systemy operacyjne mogą następnie adresować dyski twarde przy użyciu tych numerów bloków — jest to proces zwany *adresowaniem bloków logicznych* (*LBA*) — natomiast dysk twardy wewnętrznie tłumaczy numer bloku na poprawny adres CHS[5]. Chociaż nie ma na to gwarancji, odwzorowanie blok–CHS jest raczej sekwencyjne: istnieje duże prawdopodobieństwo, że blok fizyczny o numerze `n` będzie rzeczywiście graniczyć na dysku z fizycznym blokiem `n + 1`. Jest to istotne odwzorowanie sekwencyjne, co wkrótce zostanie udowodnione.

Tymczasem systemy plików istnieją jedynie w oprogramowaniu. Używają one własnych jednostek, znanych pod nazwą *bloków logicznych* (czasem nazywanych *blokami systemu plików* lub, trochę dezorientująco, po prostu *blokami*). Rozmiar bloku logicznego musi być całkowitą wielokrotnością rozmiaru bloku fizycznego. Inaczej mówiąc, blok logiczny systemu plików zawiera w sobie jeden lub więcej fizycznych bloków dyskowych.

[5] W ciągu ostatnich lat limity, dotyczące rozmiaru bloku, w dużym stopniu odpowiedzialne były za różne ograniczenia całkowitych pojemności dysków.

Działanie zarządcy operacji wejścia i wyjścia

Zarządcy operacji wejścia i wyjścia wykonują dwie podstawowe operacje: scalanie i sortowanie. *Scalanie* jest procesem polegającym na pobieraniu dwóch lub więcej sąsiadujących ze sobą zleceń operacji wejścia i wyjścia oraz na łączeniu ich, tworząc ostatecznie pojedyncze żądanie. Rozważmy dwa zlecenia: jedno dotyczące odczytu z dysku bloku o numerze 5, drugie dotyczące odczytu z dysku bloków o numerach od szóstego do siódmego. Te zlecenia mogą zostać scalone w jedno żądanie, nakazujące wykonanie operacji odczytu bloków o numerach od piątego do siódmego. Całkowita liczba operacji wejścia i wyjścia jest taka sama, lecz liczba żądań została zmniejszona o połowę.

Sortowanie to ważniejsza czynność z dwóch wspominanych operacji — jest to proces, który umieszcza w porządku rosnącym oczekujące zlecenia operacji wejścia i wyjścia, biorąc pod uwagę numery bloków. Na przykład, w przypadku operacji wejścia i wyjścia dla bloków 52, 109 oraz 7 zarządca posortuje te zlecenia w kolejności 7, 52 i 109. Jeśli następnie wykonano dodatkowe zlecenie dla bloku 81, zostanie ono wstawione pomiędzy zlecenia dla bloków 52 i 109. Zarządca operacji wejścia i wyjścia wyśle ostatecznie te zlecenia na dysk w takiej kolejności, w jakiej znajdują się one w kolejce: 7, 52, następnie 81 i w końcu 109.

W ten sposób minimalizuje się przesunięcia głowicy dysku. Zamiast wykonywania ewentualnych przypadkowych ruchów — tam i z powrotem, po całym dysku — głowica przesuwa się w sposób liniowy i płynny. Dzięki temu uzyskuje się poprawę wydajności, gdyż wyszukiwanie jest najbardziej kosztownym elementem dyskowych operacji wejścia i wyjścia.

Wspomaganie odczytów

Każda operacja odczytu musi zwrócić aktualne dane. Dlatego też, jeśli wymagane dane nie znajdują się w buforze stron, proces czytający musi zostać zablokowany, dopóki dane nie zostaną odczytane z dysku. Może to być potencjalnie długotrwałą operacją. Ten wpływ na wydajność zwany jest *opóźnieniem odczytu*.

Typowa aplikacja może rozpocząć wiele operacji wejścia i wyjścia w krótkim czasie. Ponieważ każde zlecenie jest indywidualnie zsynchronizowane, wykonanie późniejszych żądań zależy od zakończenia wcześniejszych. Rozważmy czytanie wszystkich plików z katalogu. Aplikacja otwiera pierwszy plik, wykonuje operację odczytu pakietu danych, czeka na dane, wykonuje kolejną operację odczytu itd., aż do momentu, gdy cały plik zostaje przeczytany. Następnie aplikacja rozpoczyna wykonywanie tego samego procesu dla kolejnego pliku. Żądania zostają uszeregowane: kolejne zlecenie nie może zostać wysłane, dopóki nie wykona się aktualne.

Stoi to w opozycji do działań żądań zapisu, które (w domyślnym, niezsynchronizowanym trybie) przez pewien czas nie rozpoczynają żadnych dyskowych operacji wejścia i wyjścia. Dlatego też, z perspektywy aplikacji z przestrzeni użytkownika, operacja zapisu wymaga istnienia *strumienia*, którego działanie jest niezależne od wydajności dysku. Ten strumieniowy sposób działania komplikuje problem odczytów: działający strumień zapisów może całkowicie „pochłonąć uwagę" jądra i dysku. Zjawisko to znane jest pod nazwą *zablokowania odczytów przez zapisy*.

Jeśli zarządca operacji wejścia i wyjścia *zawsze* umieszczałby nowe zlecenia w takich miejscach kolejki, by uzyskać właściwe posortowanie, możliwe byłoby trwałe zablokowanie żądań dotyczących odległych bloków. Weźmy pod uwagę poprzedni przykład. Jeśli nowe żądania byłyby wysyłane ciągle do bloków o numerach zbliżonych do 50, wówczas zlecenie dotyczące bloku 109

nigdy by się nie wykonało. Ponieważ opóźnienie odczytu jest wartością krytyczną, zachowanie to mogłoby bardzo pogorszyć wydajność systemu. Dlatego też zarządcy operacji wejścia i wyjścia implementują mechanizm chroniący przed trwałym zablokowaniem.

Proste rozwiązanie — na przykład, wzorowane na zarządcy operacji wejścia i wyjścia (o nazwie *Linus Elevator*[6]) dla wersji 2.4 jądra Linuksa — polega na tym, że następuje zatrzymanie procesu sortowania, gdy w kolejce znajduje się wystarczająco „stare" zlecenie. Powoduje to bardziej sprawiedliwe obsługiwanie żądań kosztem pogorszenia ogólnej wydajności oraz, w przypadku odczytów, poprawę współczynnika opóźnienia. Problemem pozostaje to, że ta heurystyka jest zbyt uproszczona. Dlatego też w wersji 2.6 jądra Linuksa zarządca *Linus Elevator* przestał być wspierany, a zamiast niego udostępniono kilku nowych zarządców operacji wejścia i wyjścia.

Zarządca z terminem nieprzekraczalnym

Zarządca z terminem nieprzekraczalnym (ang. *Deadline I/O Scheduler*) został zaprojektowany, aby rozwiązać problemy istniejące dla zarządcy operacji wejścia i wyjścia w wersji 2.4 jądra Linuksa oraz ogólnie usprawnić tradycyjne algorytmy wind. *Linus Elevator* zarządza posortowaną listą oczekujących zleceń operacji wejścia i wyjścia. Żądanie operacji wejścia i wyjścia znajdujące się na początku kolejki zostanie obsłużone w pierwszej kolejności. Zarządca z terminem nieprzekraczalnym również posiada taką listę, lecz znacznie usprawnia swoje działanie poprzez wprowadzenie dwóch dodatkowych kolejek: *kolejki FIFO dla odczytów* oraz *kolejki FIFO dla zapisów*. Elementy w każdej z tych kolejek są sortowane według czasu dostarczania (najpóźniej umieszczony element zostanie najwcześniej obsłużony). Kolejka FIFO dla odczytów, jak wskazuje jej nazwa, zawiera tylko żądania odczytów i posiada termin ważności 500 milisekund. Kolejka FIFO dla zapisów zawiera wyłącznie żądania zapisów i posiada termin ważności 5 sekund.

Gdy dostarczone zostaje nowe żądanie operacji wejścia i wyjścia, jest ono wstawiane we właściwe miejsce w standardowej kolejce oraz umieszczane na końcu odpowiedniej kolejki FIFO (dla odczytów lub zapisów). Zazwyczaj dysk twardy otrzymuje żądania operacji wejścia i wyjścia, wysyłane z początku standardowej, posortowanej kolejki. Poprawia to ogólną przepustowość dzięki minimalizacji liczby wyszukiwań, ponieważ zwykła kolejka posortowana jest względem numerów bloków (tak jest w przypadku zarządcy *Linus Elevator*).

Gdy element znajdujący się na początku jednej z kolejek FIFO staje się starszy, niż wynosi termin ważności związany z tą kolejką, zarządca operacji wejścia i wyjścia przerywa wysyłanie zleceń ze standardowej kolejki i rozpoczyna obsługę żądań z tej kolejki FIFO. Zlecenie znajdujące się na początku kolejki FIFO zostaje obsłużone. Oprócz tego zostaje wykonanych parę dodatkowych operacji, aby uzyskać dobry wynik. Zarządca operacji wejścia i wyjścia musi sprawdzać i obsługiwać wyłącznie zlecenia, które znajdują się na początku tej kolejki, ponieważ są one żądaniami najstarszymi.

W ten sposób zarządca *Deadline I/O Scheduler* może egzekwować nierygorystyczne przestrzeganie terminu nieprzekraczalnego dla zleceń operacji wejścia i wyjścia. Chociaż nie zapewnia to całkowicie, że żądanie operacji wejścia i wyjścia zostanie obsłużone przed upływem jego nieprzekraczalnego terminu, mimo to zarządca obsługuje zlecenia zbliżone wiekiem do terminu

[6] Tak, zarządca operacji wejścia i wyjścia został nazwany imieniem twórcy Linuksa. Zarządcy operacji wejścia i wyjścia często zwani są *algorytmami windy*, ponieważ rozwiązują problemy podobne do zapewniania płynnego działania wind.

ważności. Dlatego też zarządca z terminem nieprzekraczalnym w dalszym ciągu umożliwia uzyskanie dobrej przepustowości bez jednoczesnego blokowania pewnych żądań przez niedopuszczalnie długi czas. Ponieważ żądania odczytu posiadają krótszy termin ważności, problem blokowania odczytów przez zapisy zostaje zminimalizowany.

Zarządca przewidujący

Sposób działania zarządcy z terminem nieprzekraczalnym jest dobry, lecz nie idealny. Wróćmy do rozważań na temat zależności odczytów. W przypadku zarządcy z terminem nieprzekraczalnym pierwsze żądanie z serii kilku odczytów zostaje szybko obsłużone, przed lub w momencie upływania terminu ważności. Następnie zarządca operacji wejścia i wyjścia powraca do obsługi zleceń z kolejki posortowanej. Jak dotąd wszystko działa poprawnie. Wyobraźmy sobie jednak aplikację, która nagle „zaatakuje" system kolejnym żądaniem odczytu. W końcu nadejdzie również jego termin ważności, a zarządca operacji wejścia i wyjścia wyśle go na dysk, który wykona wyszukiwanie, by obsłużyć to żądanie, a następnie przywróci poprzednie położenie głowicy, aby kontynuować obsługę żądań z kolejki posortowanej. Takie operacje wyszukiwania po dysku mogą trwać przez jakiś czas, ponieważ wiele aplikacji zachowuje się w ten sposób. Pomimo że opóźnienie jest minimalne, globalna przepustowość nie jest zbyt dobra, gdyż żądania odczytu przychodzą przez cały czas, a dysk musi wykonywać operacje wyszukiwania, aby je obsłużyć. Wydajność mogłaby zostać poprawiona, gdyby dysk po prostu zatrzymywał się i oczekiwał na kolejny odczyt, a nie ciągle wracał, aby obsłużyć posortowaną kolejkę. Niestety, w czasie, gdy aplikacja zostaje aktywowana i wysyła swoje następne warunkowe żądanie odczytu, zarządca operacji wejścia i wyjścia właśnie „zmienił bieg na wyższy".

Problem ten ponownie wynika z dokuczliwej zależności odczytów — każde nowe żądanie odczytu zostaje wysłane tylko wtedy, gdy poprzednie się zakończyło, lecz w tym czasie, gdy aplikacja otrzymuje odczytane dane, zostaje ona aktywowana i wysyła swoje następne żądanie odczytu, natomiast zarządca operacji wejścia i wyjścia już przechodzi do dalszych zadań i zajmuje się obsługą innych zleceń. Wynikiem tego jest stracony czas na wykonanie dwóch operacji wyszukiwania dla każdego odczytu: dysk przeprowadza czynność wyszukiwania, by odczytać dane, obsługuje operację odczytu, a następnie ponownie wykonuje wyszukiwanie, wracając głowicą z powrotem. Gdyby tylko istniał jakiś sposób, aby zarządca operacji wejścia i wyjścia mógł wiedzieć —*przewidzieć*, że wkrótce zostanie wykonana inna operacja odczytu dla tego samego fragmentu dysku, wówczas czekałby, przewidując wykonanie następnego zlecenia, zamiast przesuwać głowicę tam i z powrotem. Pominięcie tych kłopotliwych wyszukiwań na pewno warte byłoby poświęcenia dodatkowych paru milisekund na oczekiwanie.

W taki właśnie sposób działa *zarządca przewidujący* (ang. *Anticipatory I/O Scheduler*). Jego poprzednikiem był zarządca z terminem nieprzekraczalnym, do którego dodano mechanizm przewidywania. Gdy zostaje wysłane żądanie odczytu, zarządca przewidujący obsługuje je jak dotychczas, przy użyciu mechanizmu nieprzekraczalnego terminu. Następnie, w przeciwieństwie do swojego poprzednika, zarządca przewidujący przestaje wykonywać jakiekolwiek czynności i przechodzi w stan oczekiwania, który może trwać do sześciu milisekund. Istnieje duże prawdopodobieństwo, że w ciągu tych sześciu milisekund aplikacja wyśle kolejne zlecenie odczytu dotyczące tego samego obszaru w systemie plików. Jeśli tak się dzieje, wówczas żądanie zostaje natychmiast obsłużone, a zarządca przewidujący ponownie przechodzi na krótko w stan oczekiwania. Jeśli sześć milisekund oczekiwania upłynie bez otrzymania jakiegokolwiek żądania odczytu, zarządca przewidujący stwierdza, że „się pomylił" i powraca do czynności, które wykonywał przedtem (to znaczy, do obsługi standardowej kolejki posortowanej). Jeśli

nawet przewidziana zostanie przeciętna liczba żądań, to i tak oszczędzi się sporo czasu (dla każdego przypadku po dwa kosztowne wyszukiwania). Ponieważ większość odczytów jest zależna od siebie, przewidywanie oszczędza mnóstwo czasu.

Zarządca ze sprawiedliwym szeregowaniem

Zarządca ze sprawiedliwym szeregowaniem (ang. *Complete Fair Queuing I/O Scheduler*, inaczej zwany *zarządcą CFQ*), podczas swojej pracy osiąga podobne rezultaty, aczkolwiek przy użyciu innej metody[7]. W tym przypadku każdy proces posiada swoją własną kolejkę, a każdej kolejce przypisany jest przedział czasowy. Zarządca ze sprawiedliwym szeregowaniem obsługuje każdą kolejkę za pomocą algorytmu cyklicznego (ang. *round-robin*), wysyłając z niej żądania dopóki, dopóty nie wyczerpie się przedział czasowy lub nie będzie już zleceń do obsłużenia. W tym drugim przypadku zarządca ze sprawiedliwym szeregowaniem przejdzie następnie na pewien czas (domyślnie 10 milisekund) w stan bezczynności i będzie oczekiwać na pojawienie się nowych żądań w kolejce. Jeśli opłaci się zastosować przewidywanie, zarządca operacji wejścia i wyjścia nie będzie wykonywać wyszukiwań. Jeśli przewidywanie nie będzie opłacalne, oczekiwanie okaże się bezcelowe i zarządca przejdzie do obsługi kolejki następnego procesu.

W każdej kolejce procesu żądania zsynchronizowane (takie jak odczyty) posiadają pierwszeństwo obsługi przed żądaniami niezsynchronizowanymi. W ten sposób zarządca CFQ faworyzuje odczyty i chroni przed wystąpieniem problemu zablokowania odczytów przez zapisy. Z powodu możliwości ustalania parametrów kolejek dla każdego procesu, zarządca CFQ sprawiedliwie obsługuje wszystkie procesy, a jednocześnie umożliwia uzyskanie dobrej wydajności ogólnej.

Zarządca ze sprawiedliwym szeregowaniem jest właściwie dopasowany do większości obciążeń i bardzo dobrze funkcjonuje jako oprogramowanie pierwszego wyboru.

Zarządca niesortujący

Zarządca niesortujący (ang. *Noop I/O Scheduler*) jest najprostszym z dostępnych zarządców. Nie wykonuje on sortowania, lecz jedynie podstawowe scalanie. Jest używany w specjalizowanych urządzeniach, które nie wymagają (lub nie przeprowadzają) własnych żądań sortowania.

Wybór i konfiguracja zarządcy operacji wejścia i wyjścia

Istnieje możliwość wyboru domyślnego zarządcy operacji wejścia i wyjścia w momencie startowania systemu, poprzez użycie parametru jądra *iosched*, podanego w linii poleceń. Dopuszczalnymi wartościami tego parametru są: *as*, *cfq*, *deadline* i *noop*. Zarządca operacji wejścia i wyjścia może zostać również wybrany dla każdego urządzenia w trakcie działania systemu, poprzez użycie pliku */sys/block/device/queue/scheduler*, gdzie `device` jest urządzeniem blokowym, dla którego należy dokonać wyboru. Odczyt tego pliku zwraca nazwę aktualnego zarządcy operacji wejścia i wyjścia; zapis jednej z możliwych opcji do tego pliku spowoduje wybranie danego zarządcy. Na przykład, aby dla urządzenia *hda* wybrać zarządcę CFQ, należy wykonać następującą operację:

```
# echo cfq > /sys/block/hda/queue/scheduler
```

[7] W tym podrozdziale przeprowadzona zostaje analiza aktualnej implementacji zarządcy ze sprawiedliwym szeregowaniem. W poprzednich wersjach tego zarządcy nie były używane przedziały czasowe ani heurystyka przewidywania, lecz mimo tego działały one w podobny sposób.

Katalog */sys/block/device/queue/iosched* zawiera pliki, które pozwalają administratorowi na odzyskiwanie i ustawianie parametrów służących do dostrajania działania danego zarządcy operacji wejścia i wyjścia. Dokładne wartości parametrów zależą od aktualnego zarządcy. Zmiana dowolnego z tych parametrów wymaga uprawnień administratora.

Dobry programista tworzy programy, które nie są związane z podstawowym podsystemem wejścia i wyjścia. Mimo to wiedza na temat tego podsystemu może pomóc w tworzeniu zoptymalizowanego kodu.

Optymalizowanie wydajności operacji wejścia i wyjścia

Ponieważ operacje dyskowe wejścia i wyjścia są bardzo wolne w porównaniu z wydajnością innych składników systemu, a jednocześnie stanowią bardzo ważny element nowoczesnej techniki komputerowej, dlatego też poprawa ich wydajności ma kluczowe znaczenie.

Podczas programowania systemowego należy zawsze rozważyć zastosowanie takich opcji jak zmniejszanie liczby operacji wejścia i wyjścia (poprzez łączenie wielu niedużych operacji w kilka większych), wykonywanie operacji wejścia i wyjścia wyrównanych do rozmiaru bloku lub używanie buforowania w przestrzeni użytkownika (opisanego w rozdziale 3.) oraz wykorzystywanie zaawansowanych technik, takich jak techniki wektorowe, pozycyjne (opisane w rozdziale 2.) oraz asynchroniczne operacje wejścia i wyjścia.

W aplikacjach najbardziej wymagających, kluczowych i intensywnie używających operacji wejścia i wyjścia, można jednak zastosować dodatkowe sposoby poprawy ich wydajności. Chociaż jądro Linuksa, jak wcześniej napisano, używa zaawansowanych zarządców operacji wejścia i wyjścia, aby zminimalizować liczbę niepożądanych wyszukiwań dyskowych, aplikacje z przestrzeni użytkownika mogą również w podobny sposób dążyć do tego samego celu, by jeszcze bardziej poprawić wydajność.

Szeregowanie operacji wejścia i wyjścia w przestrzeni użytkownika

W aplikacjach intensywnie używających i generujących dużą liczbę żądań operacji wejścia i wyjścia należy zwracać szczególną uwagę na poprawę ich wydajności i dlatego też można dla nich implementować sortowanie i scalanie oczekujących zleceń, dzięki czemu będą one wykonywać te same funkcje jak zarządca operacji wejścia i wyjścia dla Linuksa[8].

Dlaczego jednak wykonywać tę samą pracę dwukrotnie, jeśli wie się, że zarządca operacji wejścia i wyjścia inteligentnie posortuje żądania, minimalizując liczbę przeprowadzonych wyszukiwań oraz pozwalając głowicy dysku na poruszanie się w sposób płynny i liniowy? Rozważmy przykład aplikacji, która wysyła dużą liczbę nieposortowanych żądań operacji wejścia i wyjścia. Żądania te pojawiają się w kolejce zarządcy operacji wejścia i wyjścia w porządku losowym. Zarządca wykonuje swoją funkcję poprzez sortowanie i łączenie żądań, zanim zostaną one wysłane na dysk, ale żądania zaczynają docierać do dysku już wtedy, gdy aplikacja w dalszym ciągu zajęta jest generowaniem operacji wejścia i wyjścia oraz wysyłaniem nowych zleceń. Zarządca operacji wejścia i wyjścia może w danym momencie posortować jedynie niewielki

[8] Opisane tu techniki powinny być stosowane tylko dla aplikacji kluczowych i intensywnie używających operacji wejścia i wyjścia. Sortowanie żądań operacji wejścia i wyjścia (przy założeniu, że istnieją dane do posortowania) dla aplikacji, które nie generują zbyt dużo takich zleceń, jest bezsensowne i niepotrzebne.

zestaw żądań — na przykład, kilka z tej aplikacji i jakieś inne, które właśnie oczekują. Każdy pakiet żądań, wysłany z aplikacji, zostaje starannie posortowany, lecz nie są uwzględniane problemy związane z zapełnieniem kolejki czy przyszłymi zleceniami.

Dlatego też, jeśli aplikacja generuje wiele żądań — szczególnie, jeśli dotyczą one danych znajdujących się na całym dysku — można osiągnąć korzyści z wcześniejszego posortowania żądań, zanim zostaną one wysłane. Dzięki temu żądania te dotrą już uporządkowane do zarządcy operacji wejścia i wyjścia.

Aplikacja z przestrzeni użytkownika nie posiada jednak dostępu do tej samej informacji co jądro. Przy przetwarzaniu żądań w najniższych warstwach zarządcy operacji wejścia i wyjścia, operuje się fizycznymi blokami dysku. Ich sortowanie jest prostą czynnością. Lecz w przestrzeni użytkownika, podczas przetwarzania żądań operuje się nazwami plików i wartościami położeń. Aplikacje z przestrzeni użytkownika muszą wydobywać informację i inteligentnie odgadywać jak wygląda budowa systemu plików.

Mając określony cel, polegający na ustaleniu najlepszego (z punktu widzenia wyszukiwania na dysku) algorytmu sortowania listy żądań operacji wejścia i wyjścia dla określonych plików, aplikacja z przestrzeni użytkownika posiada kilka opcji wyboru. Sortowanie może być oparte na następujących rodzajach danych:

- nazwie pełnej ścieżki,
- numerze i-węzła,
- numerze fizycznego bloku dysku dla danego pliku.

Każda z tych opcji sortowania zawiera w sobie rozwiązanie kompromisowe. Poniżej przedstawiono ich krótki przegląd.

Sortowanie wg ścieżki. Sortowanie według nazwy ścieżki jest najprostszą, choć najmniej wydajną metodą ustalania fizycznego położenia plików. Z powodu działania algorytmów rozmieszczania, używanych w większości systemów plików, pliki w każdym katalogu (jak też katalogi, posiadające wspólny katalog nadrzędny) są rozmieszczane obok siebie na dysku. Prawdopodobieństwo, że pliki znajdujące się w tym samym katalogu zostały stworzone w przybliżeniu w tym samym czasie, jedynie wzmacnia tę charakterystykę rozmieszczania.

Dlatego też sortowanie według ścieżki w ogólny sposób ustala fizyczne położenia plików na dysku. Prawdą jest, że dwa pliki znajdujące się w tym samym katalogu mają większe szanse znajdować się obok siebie, niż dwa inne pliki z radykalnie odmiennych fragmentów systemu plików. Minusem takiego rozwiązania jest to, że nie obejmuje ono swoim działaniem tematu fragmentacji: im bardziej system plików jest sfragmentowany, tym mniej użyteczne staje się sortowanie według ścieżki. Nawet podczas ignorowania fragmentacji, sortowanie według ścieżki tylko przybliża rzeczywiste uporządkowanie według numerów bloków. Plusem sortowania wg ścieżki jest to, iż w jakimś stopniu nadaje się ono do zastosowania we wszystkich systemach plików. Niezależnie od podejścia do problemu ustalenia położenia pliku uwzględnienie tymczasowej lokalizacji pozwala stwierdzić, że sortowanie według ścieżki będzie rozwiązaniem co najmniej średnio dokładnym. Ten rodzaj sortowania można również przeprowadzić w prosty sposób.

Sortowanie wg numeru i-węzła. I-węzły są konstrukcjami uniksowymi zawierającymi metadane, związane z poszczególnymi plikami. Podczas gdy dane pliku mogą zajmować wiele fizycznych bloków na dysku, sam plik posiada dokładnie jeden i-węzeł, który zawiera takie informacje

jak rozmiar pliku, uprawnienia, nazwa właściciela itd. I-węzły zostaną dokładniej przeanalizowane w rozdziale 7. W tym momencie wystarczy znać dwa fakty: każdy plik posiada związany z nim i-węzeł, a każdemu i-węzłowi przypisany jest unikalny numer (identyfikator).

Sortowanie według i-węzła jest lepsze niż sortowanie wg ścieżki, przy założeniu następującej relacji:

```
numer i-węzła pliku i < numer i-węzła pliku j
```

Wynika z tego, że:

```
numery bloków fizycznych pliku i < numery bloków fizycznych pliku j
```

Jest to oczywiście prawdą w przypadku takich systemów plików Uniksa, jak *ext2* czy *ext3*. Dla systemów plików, które nie używają pojęcia i-węzła, istnieje również jakieś rozwiązanie, lecz uwzględnienie i-węzłów (bez względu na to, na co są one odwzorowane) pozwala w dalszym ciągu na uzyskanie dobrego przybliżenia.

Numer i-węzła można otrzymać przy użyciu funkcji systemowej `stat()`, również omówionej w rozdziale 7. Jeśli uzyska się dostęp do numeru i-węzła dla każdego pliku, który brany jest pod uwagę podczas wykonywania żądań operacji wejścia i wyjścia, zlecenia te mogą zostać posortowane w porządku rosnącym wg tego identyfikatora.

Oto przykład prostego programu, który wypisuje numer i-węzła dla danego pliku:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/*
* get_inode — zwraca numer i-węzła dla pliku związanego
* z podanym deskryptorem pliku lub -1 w przypadku błędu
*/
int get_inode (int fd)
{
   struct stat buf;
   int ret;

   ret = fstat (fd, &buf);
   if (ret < 0)
   {
      perror ("fstat");
      return -1;
   }
   return buf.st_ino;
}

int main (int argc, char *argv[])
{
   int fd, inode;
   if (argc < 2)
   {
      fprintf (stderr, "Użycie programu: %s <plik>\n", argv[0]);
      return 1;
   }

   fd = open (argv[1], O_RDONLY);
   if (fd < 0)
   {
      perror ("open");
```

```
        return 1;
    }

    inode = get_inode (fd);
    printf ("%d\n", inode);
    return 0;
}
```

Funkcję `get_inode()` można łatwo zaadaptować do użycia we własnych aplikacjach.

Sortowanie według numeru i-węzła posiada kilka zalet: numer i-węzła można łatwo uzyskać, można takie numery w prosty sposób posortować, a sortowanie jest również dobrym przybliżeniem położenia pliku fizycznego. Główne minusy tego rozwiązania to: fragmentacja powoduje pogorszenie przybliżenia, a samo przybliżenie jest mniej dokładne w przypadku nieuniksowych systemów plików. Bez względu na to jest to najczęściej używane rozwiązanie dla szeregowania żądań operacji wejścia i wyjścia w przestrzeni użytkownika.

Sortowanie wg numeru fizycznego bloku. Najlepszym wariantem podczas tworzenia własnego algorytmu windy jest oczywiście sortowanie wg numeru fizycznego bloku dysku. Jak już wcześniej napisano, każdy plik jest podzielony na bloki logiczne, które są najmniejszymi jednostkami alokacji w systemie plików. Rozmiar bloku logicznego zależy od rodzaju systemu plików: każdy blok logiczny jest odwzorowany na pojedynczy blok fizyczny. Dlatego też można uzyskać numery bloków logicznych dla danego pliku, a następnie określić, na jakie bloki fizyczne są one odwzorowane i w końcu posortować te wartości.

Jądro dostarcza metody, pozwalającej uzyskać numer fizycznego bloku dysku na podstawie numeru logicznego bloku na dysku. Jest to realizowane za pomocą funkcji systemowej `ioctl()` (omówionej w rozdziale 7.), która służy do wykonania polecenia `FIBMAP`:

```
ret = ioctl (fd, FIBMAP, &block);
if (ret < 0)
    perror ("ioctl");
```

Parametr `fd` jest deskryptorem pliku, natomiast `block` jest numerem bloku logicznego, dla którego należy określić blok fizyczny. Po wykonaniu funkcji, w przypadku sukcesu parametr `block` zawiera numer bloku fizycznego. Przekazywane numery bloków logicznych rozpoczynają się od zera i obejmują dany plik. Oznacza to, że w przypadku, gdy plik składa się z ośmiu bloków logicznych, poprawnymi wartościami ich numerów są liczby od zera do siedem.

Ustalanie parametrów odwzorowania bloku logicznego na fizyczny jest więc procesem dwuetapowym. Po pierwsze, należy określić liczbę bloków dla danego pliku. Realizowane jest to poprzez użycie funkcji systemowej `stat()`. Po drugie, dla każdego bloku logicznego należy wykonać funkcję `ioctl()`, która pozwoli na uzyskanie numeru dla odpowiedniego bloku fizycznego.

Oto prosty program wykonujący powyższą operację dla pliku, którego nazwa podawana jest w linii poleceń:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <linux/fs.h>
```

```
/*
* get_block — zwraca odwzorowanie bloku fizycznego dla bloku logicznego
* dla pliku związanego z podanym deskryptorem pliku fd
*/
int get_block (int fd, int logical_block)
{
   int ret;

   ret = ioctl (fd, FIBMAP, &logical_block);
   if (ret < 0)
   {
      perror ("ioctl");
      return -1;
   }

   return logical_block;
}

/*
* get_nr_blocks — zwraca liczbę bloków logicznych
* używanych przez plik, związany z deskryptorem pliku fd
*/
int get_nr_blocks (int fd)
{
   struct stat buf;
   int ret;

   ret = fstat (fd, &buf);
   if (ret < 0)
   {
      perror ("fstat");
      return -1;
   }

   return buf.st_blocks;
}

/*
* print_blocks — dla każdego bloku logicznego używanego przez plik
* związany z deskryptorem pliku fd, wypisuje na standardowe wyjście
* parę wartości: "(numer bloku logicznego, numer bloku fizycznego)"
*/
void print_blocks (int fd)
{
   int nr_blocks, i;

   nr_blocks = get_nr_blocks (fd);
   if (nr_blocks < 0)
   {
      fprintf (stderr, "Wywołanie funkcji get_nr_blocks nie powiodło się!\n");
      return;
   }
   if (nr_blocks == 0)
   {
      printf ("Nie ma przydzielonych bloków\n");
      return;
   }
   else if (nr_blocks == 1)
      printf ("1 blok\n\n");
   else
      printf ("%d bloki(ów)\n\n", nr_blocks);

   for (i = 0; i < nr_blocks; i++)
   {
```

```
        int phys_block;

        phys_block = get_block (fd, i);
        if (phys_block < 0)
        {
           fprintf (stderr, "Wywołanie funkcji get_block nie powiodło się!\n");
           return;
        }
        if (!phys_block)
           continue;

        printf ("(%u, %u) ", i, phys_block);
     }

     putchar ('\n');
  }

  int main (int argc, char *argv[])
  {
     int fd;

     if (argc < 2)
     {
        fprintf (stderr, "Użycie programu: %s <plik>\n", argv[0]);
        return 1;
     }

     fd = open (argv[1], O_RDONLY);
     if (fd < 0)
     {
        perror ("open");
        return 1;
     }

     print_blocks (fd);

     return 0;
  }
```

Ponieważ pliki raczej przylegają do siebie, a operacja sortowania żądań operacji wejścia i wyjścia wg numerów bloków logicznych jest trudna (w najlepszym razie) do zrealizowania, sensowne wydaje się sortowanie oparte na położeniu tylko pierwszego bloku logicznego dla danego pliku. Zgodnie z tym, funkcja `get_nr_blocks()` nie jest wymagana, a powyższy program może wykonać sortowanie oparte na wartości zwracanej przez poniższą funkcję:

```
get_block (fd, 0);
```

Wadą rozwiązania polegającego na użyciu polecenia `FIBMAP` jest to, iż wymaga ono uprawnienia `CAP_SYS_RAWIO` — faktycznie oznacza to posiadanie uprawnienia administratora. Dlatego też aplikacje, które nie zostały uruchomione przez administratora, nie mogą używać tej metody. Poza tym, mimo że polecenie `FIBMAP` zdefiniowane jest w standardzie, to jego rzeczywista implementacja zależy od danego systemu plików. Powszechnie używane systemy plików, takie jak *ext2* lub *ext3* wspierają to polecenie, natomiast bardziej zaawansowane mogą tego nie robić. Funkcja systemowa `ioctl()` zwraca wartość `EINVAL`, gdy polecenie `FIBMAP` nie jest wspierane.

Zaletą tego rozwiązania jest chociażby to, że udostępnia ono dla danego pliku *rzeczywisty* numer jego bloku fizycznego, który jest wartością potrzebną do procesu sortowania. Nawet jeśli posortuje się wszystkie operacje wejścia i wyjścia dla danego pliku, opierając się na położeniu tylko jednego bloku (zarządca operacji wejścia i wyjścia dla jądra sortuje każde pojedyncze żądanie

w oparciu o numer bloku), rozwiązanie to pozwoli uzyskać wyniki bardzo przybliżone do optymalnego szeregowania. Wymaganie posiadania uprawnienia administratora jest jednak opcją, która nie może być zrealizowana w przypadku wielu aplikacji.

Zakończenie

W trzech ostatnich rozdziałach książki przedstawiono wszystkie aspekty działania plikowych operacji wejścia i wyjścia w Linuksie. W rozdziale 2. omówiono podstawy plikowych operacji wejścia i wyjścia w Linuksie (w rzeczywistości podstawy programowania w Uniksie), uwzględniając przy tym takie funkcje systemowe jak `read()`, `write()`, `open()` i `close()`. W rozdziale 3. przeanalizowano buforowanie w przestrzeni użytkownika oraz jego implementację w standardowej bibliotece języka C. W niniejszym rozdziale przedstawiono aspekty zaawansowanych operacji wejścia i wyjścia, poczynając od bardziej potężnych, lecz jednocześnie bardziej skomplikowanych funkcji systemowych, a kończąc na technikach optymalizacyjnych oraz niszczącym wydajność wyszukiwaniu po dysku.

W kolejnych dwóch rozdziałach omówione zostanie zarządzanie procesami: tworzenie, usuwanie i kontrolowanie procesów.

ROZDZIAŁ 5.

Zarządzanie procesami

Jak już wspomniano w rozdziale 1., podstawowymi abstrakcjami w systemie Unix oprócz plików są procesy. Procesy są wykonywanymi kodami obiektów, czyli aktywnymi, „żyjącymi", działającymi programami. Proces to jednak coś więcej niż tylko kod maszynowy — składa się on z danych, zasobów, stanu procesu oraz ze zwirtualizowanego komputera.

W tym rozdziale omówione zostaną podstawy zarządzania procesami, poczynając od ich tworzenia, a kończąc na usuwaniu. Te podstawy nie zmieniły się wiele od początków istnienia systemu Unix. Wyraźnie widać, że ciągłość idei, a także myślenie przyszłościowe, zastosowane podczas projektowania systemu Unix, znajdują swoje potwierdzenie podczas omawiania tematów związanych z zarządzaniem procesami. Dla Uniksa wybrano rzadkie rozwiązanie i oddzielono czynność tworzenia nowego procesu od czynności ładowania nowego obrazu binarnego. Chociaż te dwa zadania wykonywane są najczęściej wspólnie, podział ten pozwolił na uzyskanie dużej wolności, pozwalającej na eksperymentowanie i dalszy rozwój każdego z nich. To rzadko stosowane rozwiązanie przetrwało po dzień dzisiejszy i choć większość systemów operacyjnych oferuje pojedynczą funkcję systemową pozwalającą na uruchomienie nowego programu, Unix wymaga do tego celu dwóch funkcji: `fork` i `exec`. Lecz zanim zostaną one omówione, należy dokładniej przyjrzeć się samym procesom.

Identyfikator procesu

Każdy proces reprezentowany jest poprzez unikalny identyfikator, zwaną *identyfikatorem procesu* (ang. *process ID*, w skrócie *PID*). Istnieje gwarancja, że identyfikator ten jest unikalny *w danym momencie*. Oznacza to, że jeśli w chwili czasu `t0` istnieje tylko jeden proces posiadający PID równy 770 (zakładając, że w ogóle będzie istniał jakiś proces o takiej wartości), to wówczas nie ma gwarancji na to, że w chwili czasu `t1` nie będzie innego procesu o takim samym PID. Zasadniczo jednak dla większości kodu istnieje założenie, że jądro niezbyt chętnie używa ponownie tych samych identyfikatorów procesów — założenie to, jak się wkrótce okaże, jest wystarczająco bezpieczne.

Proces jałowy to proces, który działa w jądrze, gdy nie istnieją inne procesy uruchamialne. Posiada on PID równy zeru. Pierwszy proces, który zostaje uruchomiony przez jądro po wystartowaniu systemu, zwany jest *procesem inicjalizującym* i posiada PID równy 1. Zazwyczaj procesem inicjalizującym w Linuksie jest program o nazwie *init*. Termin „init" stosuje się w celu zarówno

opisania procesu inicjalizującego, który uruchamiany jest przez jądro, jak i nazwania określonego programu używanego w tym celu[1].

Dopóki użytkownik nie zleci jądru, jaki program powinien zostać uruchomiony (używając do tego celu parametru linii poleceń dla programu *init*), jądro musi samo rozpoznać odpowiedni proces inicjalizujący. Jest to rzadki przypadek, kiedy jądro dyktuje strategię działania. Jądro Linuksa próbuje użyć czterech programów wykonywanych w następującej kolejności:

1. Program */sbin/init*: jest to preferowane i najbardziej prawdopodobne położenie procesu inicjalizującego.
2. Program */etc/init*: inne prawdopodobne położenie procesu inicjalizującego.
3. Program */bin/init*: możliwe położenie procesu inicjalizującego.
4. Program */bin/sh*: położenie powłoki systemowej *Bourne shell*, która zostanie uruchomiona przez jądro, jeśli nie znajdzie ono procesu inicjalizującego.

Pierwszy z tych procesów, który zostanie odnaleziony, będzie uruchomiony jako proces inicjalizujący. Jeśli uruchomienie wszystkich czterech procesów nie powiedzie się, system zostanie zatrzymany przez jądro Linuksa w trybie paniki.

Po uruchomieniu go przez jądro, proces inicjalizujący wykonuje dalszą część procesu ładowania systemu. Zwykle jest to inicjalizacja systemu, uruchomienie różnych usług oraz wywołanie powłoki logowania.

Przydział identyfikatorów procesów

Jądro domyślnie narzuca maksymalną wartość identyfikatora procesu, wynoszącą 32768. Jest to wykonywane w celu zapewnienia kompatybilności ze starszymi wersjami systemów uniksowych, które używały mniejszych, 16-bitowych typów danych, aby przechowywać identyfikatory procesów. Administratorzy systemowi mogą jednak zwiększyć tę wartość przy użyciu pliku */proc/sys/kernel/pid_max*, uzyskując dzięki temu większy zakres wartości PID kosztem zmniejszonej kompatybilności.

Jądro przydziela procesom ich identyfikatory w sposób liniowy. Jeśli PID o numerze 17 jest najwyższym aktualnie przydzielonym numerem, wówczas w następnej kolejności zostanie przydzielony PID równy 18, nawet gdyby proces posiadający identyfikator o numerze 17 nie istniał już w systemie w momencie uruchamiania nowego procesu. Jądro nie przydzieli ponownie „starych" identyfikatorów, dopóki nie zużyje wszystkich dostępnych wartości — to znaczy niższe identyfikatory nie zostaną przydzielone, dopóki nie będzie użyta wartość zapisana w pliku */proc/sys/kernel/pid_max*. Linux w przypadku dłuższych okresów nie gwarantuje unikalności dla identyfikatorów procesów. Niepowtarzalność wartości PID oraz przynajmniej krótkoterminowy komfort stabilności wynika ze sposobu ich przydzielania.

[1] Większe podobieństwo widać w oryginalnym, angielskim nazewnictwie: proces inicjalizujący to *init process*, natomiast program o nazwie *init* to *init program — przyp. tłum.*

Hierarchia procesów

Proces uruchamiający nowy proces zwany jest *rodzicem* lub *procesem rodzicielskim* (ang. *parent*); nowy proces nazywany jest *potomkiem* lub *procesem potomnym* (ang. *child*). Każdy proces (poza procesem inicjalizującym) uruchomiony zostaje przez inny proces. Dlatego też każdy proces potomny ma swój proces rodzicielski. Ta zależność zapisana zostaje dla każdego procesu w jego *identyfikatorze procesu rodzicielskiego* (PPID), który jest identyfikatorem PID rodzica dla danego potomka.

Każdy proces posiada *użytkownika* i *grupę właścicielską*. Prawo własności używane jest w celu kontrolowania uprawnień dostępu do zasobów. Użytkownicy i grupy są dla jądra jedynie liczbami całkowitymi. Poprzez pliki */etc/passwd* i */etc/group* liczby te zostają odwzorowane na przyjazne dla ludzi nazwy, znane każdemu użytkownikowi systemu Unix, takie jak konto *root* czy grupa *wheel* (jądro Linuksa nie wymaga do prawidłowej pracy żadnych łańcuchów tekstowych, które mogą być odczytywane przez ludzi; woli utożsamiać obiekty z liczbami całkowitymi). Każdy proces potomny dziedziczy prawa własności użytkownika i grupy właścicielskiej od swojego rodzica.

Każdy proces jest również elementem *grupy procesów*, która po prostu określa jego powiązania z innymi procesami i nie może być mylona z poprzednio opisaną koncepcją użytkownika i grupy. Procesy potomne zwykle należą do tej samej grupy procesów, co ich procesy rodzicielskie. Dodatkowo, gdy powłoka uruchamia przetwarzanie potokowe (np., gdy użytkownik wprowadza tekst *ls | less*), wszystkie polecenia z tego potoku znajdują się w tej samej grupie procesów. Pojęcie grupy procesów pozwala na łatwe wysyłanie sygnałów lub odbieranie informacji z całego potoku, jak również od jego wszystkich procesów potomnych. Z perspektywy użytkownika pojęcie grupy procesów jest ściśle związane z pojęciem *zadania* (ang. *job*).

Typ pid_t

Od strony programistycznej identyfikator procesu reprezentowany jest przez typ danych `pid_t`, który zdefiniowany jest w pliku nagłówkowym `<sys/types.h>`. Dokładna wartość typu pierwotnego dla języka C zależy od architektury i nie jest zdefiniowana w żadnym standardzie języka C. W Linuksie jest to jednakże alias do typu `int` języka C, utworzony za pomocą słowa kluczowego `typedef`.

Otrzymywanie identyfikatora procesu oraz identyfikatora procesu rodzicielskiego

Funkcja systemowa `getpid()` zwraca identyfikator procesu dla procesu wywołującego:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid (void);
```

Funkcja systemowa `getppid()` zwraca identyfikator procesu dla rodzica procesu wywołującego:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid (void);
```

Żadna z tych funkcji nie zwraca kodu błędu. Dlatego też ich użycie jest proste:

```
printf ("Mój PID=%d\n", getpid ( ));
printf ("PID rodzica=%d\n", getppid ( ));
```

Skąd wiadomo, że `pid_t` oznacza typ liczb całkowitych ze znakiem? Dobre pytanie! Odpowiedź jest prosta: nie wiadomo. Nawet jeśli założy się, że typ `pid_t` równy jest w Linuksie typowi `int`, takie zgadywanie wciąż zawodzi podczas ustalania typu abstrakcyjnego i pogarsza przenośność. Niestety, podobnie jak dla innych aliasów w języku Csaturn-nowy uzyskiwanych za pomocą słowa kluczowego `typedef`, nie istnieje prosty sposób na wyświetlenie wartości zmiennych o typie `pid_t` — jest to fragment abstrakcji i technicznie rzecz biorąc, potrzebna w tym celu byłaby funkcja `pid_to_int()`, która nie istnieje. Zakładanie, że te wartości są liczbami całkowitymi, jest jednak powszechnym zachowaniem, przynajmniej dla potrzeb funkcji `printf()`.

Uruchamianie nowego procesu

Czynność ładowania do pamięci i uruchamiania obrazu programu oddzielona jest w Uniksie od czynności tworzenia nowego procesu. Jedna funkcja systemowa (w rzeczywistości, jedna funkcja z grupy) ładuje program binarny do pamięci, zastępując poprzednią zawartość przestrzeni adresowej, oraz rozpoczyna działanie nowego programu. Czynność ta zwana jest *uruchomieniem* (ang. *executing*) nowego programu, a jej wykonanie umożliwia rodzina funkcji *exec*.

Inna funkcja systemowa służy do utworzenia nowego procesu, który na początku jest prawie kopią swojego procesu rodzicielskiego. Nowy proces często wywołuje nowy program. Czynność tworzenia nowego procesu zwana jest w języku Uniksa *rozwidleniem* lub *pomnożeniem* (ang. *forking*) i jest realizowana za pomocą funkcji systemowej `fork()`. Dwie czynności — najpierw rozwidlenie, aby utworzyć nowy proces, a następnie uruchomienie, by załadować nowy obraz dla tego procesu — są więc wymagane, aby uruchomić obraz nowego programu w nowym procesie. Najpierw zostanie omówiona rodzina funkcji *exec*, a następnie poddana analizie zostanie funkcja `fork()`.

Rodzina funkcji exec

Nie istnieje pojedyncza funkcja *exec*; mamy do czynienia z rodziną funkcji *exec*, utworzoną na podstawie jednej funkcji systemowej. Oto najprostsza z tych funkcji, zwana `execl()`:

```
#include <unistd.h>

int execl (const char *path, const char *arg, ...);
```

Wywołanie funkcji `execl()` zastępuje obraz aktualnego procesu nowym obrazem poprzez załadowanie do pamięci programu, którego położenie wskazywane jest przez parametr `path`. Parametr `arg` staje się pierwszym argumentem tego programu. Znak wielokropka oznacza zmienną liczbę parametrów — funkcja `execl()` jest funkcją o zmiennej liczbie parametrów (ang. *variadic*), co oznacza, że może posiadać więcej dodatkowych parametrów. Lista parametrów musi być zakończona wartością `NULL`.

Na przykład, poniższy kod zastępuje aktualnie działający program przez */bin/vi*:

```
int ret;

ret = execl ("/bin/vi", "vi", NULL);
```

```
if (ret == -1)
   perror ("execl");
```

Należy zauważyć, że postąpiono zgodnie z konwencją Uniksa i jako pierwszy parametr programu przekazano łańcuch „vi". Powłoka systemowa przekazuje ostatni element ścieżki, którym w tym przypadku jest łańcuch „vi", jako pierwszy parametr dla uruchomionego lub rozwidlonego procesu, tak więc program może sprawdzić swój pierwszy argument wywołania — `argv[0]`, aby odczytać nazwę swojego obrazu binarnego. W wielu przypadkach pewne systemowe programy użytkowe, posiadające różne nazwy, są faktycznie pojedynczym programem, do którego istnieją twarde dowiązania pozwalające na uzyskanie odmiennych nazw. Program wykorzystuje swój pierwszy parametr, aby odkryć to zachowanie.

Inny przykład kodu pozwala na edycję pliku */home/kidd/hooks.txt*:

```
int ret;

ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL);
if (ret == -1)
   perror ("execl");
```

Zazwyczaj funkcja `execl()` nie wraca do procedury wywołującej. Poprawne jej wywołanie kończy się skokiem do punktu wejściowego nowego programu, a kod, który był przedtem wykonywany, zostaje usunięty z przestrzeni adresowej procesu. Jednak w przypadku błędu funkcja `execl()` zwraca `−1` oraz odpowiednio ustawia zmienną `errno`, by opisać problem. W dalszej części tego podrozdziału omówione zostaną możliwe wartości zmiennej `errno`.

Poprawne wywołanie funkcji `execl()` zmienia nie tylko przestrzeń adresową oraz obraz procesu, lecz także pewne inne jego atrybuty:

- Dowolne oczekujące sygnały zostają usunięte.
- Dowolnym sygnałom, które są właśnie przechwytywane przez proces (przechwytywanie omówione jest w rozdziale 9.), zostaje przywrócone domyślne działanie, ponieważ procedury obsługi sygnałów nie istnieją już w przestrzeni adresowej procesu.
- Dowolne blokady pamięci (omówione w rozdziale 8.) zostają usunięte.
- Większość atrybutów wątków otrzymuje domyślne wartości.
- Większość statystyk dla procesów zostaje wyzerowanych.
- Wszystko, co związane jest z pamięcią procesu, włącznie z plikami odwzorowanymi, zostaje usunięte.
- Wszystko, co istnieje wyłącznie w przestrzeni użytkownika, włącznie z cechami biblioteki języka C, takimi jak sposób działania funkcji `atexit()`, zostaje usunięte.

Wiele właściwości procesu jednak *nie* zmienia się. Na przykład PID, PID rodzica, priorytet, właściciel oraz grupa właścicielska pozostają niezmienione.

Zazwyczaj przy wykonaniu *exec* następuje dziedziczenie plików otwartych. Oznacza to, że nowo uruchomiony program ma pełny dostęp do wszystkich otwartych plików z procesu pierwotnego, zakładając, że zna wartości deskryptorów plików. Jednakże często nie jest to pożądanym zachowaniem. Zwykle przed wykonaniem *exec* powinno się zamykać otwarte pliki, choć można też powiadomić jądro, aby przeprowadziło ten proces automatycznie poprzez odpowiednie wywołanie funkcji `fcntl()`.

Pozostałe elementy grupy funkcji exec

Oprócz samej funkcji `execl()` istnieje jeszcze pięć innych elementów wchodzących w skład grupy funkcji *exec*:

```
#include <unistd.h>

int execlp (const char *file, const char *arg, ...);

int execle (const char *path, const char *arg, ..., char * const envp[]);

int execv (const char *path, char *const argv[]);

int execvp (const char *file, char *const argv[]);

int execve (const char *filename, char *const argv[], char *const envp[]);
```

Skróty mnemoniczne są proste. Litery `l` oraz `v` oznaczają, że parametry dostarczone zostały odpowiednio w liście lub w tablicy (wektorze). Litera `p` oznacza, że w poszukiwaniu danego pliku zostanie sprawdzona cała ścieżka przeszukiwań użytkownika (zmienna środowiskowa `$PATH`). Polecenia, które używają wariantów nazw z literą `p`, mogą używać tylko nazwy pliku, jeśli katalog, w którym ten plik się znajduje, zawarty jest w ścieżce przeszukiwań użytkownika. Wreszcie litera `e` oznacza, że dla nowego procesu dostarczone zostaje nowe środowisko. Ciekawostką jest fakt, że rodzina funkcji *exec* nie zawiera żadnego elementu, który zarówno wykorzystuje ścieżkę przeszukiwań, jak również otrzymuje nowe środowisko, pomimo iż nie istnieje żadna techniczna przyczyna tego pominięcia. Wynika to prawdopodobnie stąd, że warianty z literą `p` zostały zaimplementowane dla używania z powłoką systemową, a procesy uruchamiane z poziomu powłoki zazwyczaj dziedziczą od niej swoje środowisko.

Elementy rodziny funkcji *exec*, których parametrem jest tablica, działają tak samo, jak inne funkcje, za wyjątkiem tego, że zamiast listy stworzona i przekazana zostaje tablica argumentów. Użycie tablicy pozwala na ustalanie argumentów w czasie działania programu. Podobnie jak lista parametrów o zmiennej długości, również tablica musi zostać zakończona elementem zawierającym wartość `NULL`.

Następujący fragment kodu używa funkcji `execvp()`, aby uruchomić program *vi*, tak jak to zostało wykonane w poprzednim przykładzie:

```
const char *args[] = { "vi", "/home/kidd/hooks.txt", NULL };
int ret;

ret = execvp ("vi", args);
if (ret == -1)
   perror ("execvp");
```

Przy założeniu, że katalog */bin* zawarty jest w ścieżce przeszukiwań użytkownika, ten przykład kodu będzie działać w podobny sposób jak poprzedni.

W Linuksie tylko jeden element rodziny *exec* jest funkcją systemową. Pozostałe z nich to interfejsy programowe (ang. *wrappers*) dla tej funkcji systemowej, zaimplementowane w bibliotece języka C. Tylko funkcja `execve()` jest oryginalną funkcją systemową. Istnieją dwa powody takiego rozwiązania. Po pierwsze, funkcje systemowe ze zmienną liczbą parametrów mogłyby być trudne do zaimplementowania, a po drugie, ścieżka przeszukiwań użytkownika istnieje jedynie w jego przestrzeni. Prototyp funkcji systemowej jest taki sam jak w przypadku funkcji udostępnionej użytkownikowi.

Kody błędów

W przypadku sukcesu, funkcje z rodziny *exec* nie wracają do procedury wywołującej. W przypadku błędu, zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

`E2BIG`
: Całkowita liczba bajtów dla dostarczonej listy parametrów (`arg`) lub środowiska (`envp`) jest zbyt duża.

`EACCESS`
: Proces nie posiada uprawnienia szukania elementu w ścieżce `path`; ścieżka `path` nie jest plikiem zwykłym; plik docelowy nie jest plikiem wykonywalnym; system plików, w którym znajduje się ścieżka `path` lub plik `file`, jest zamontowany w trybie `noexec`.

`EFAULT`
: Podany wskaźnik jest niepoprawny.

`EIO`
: Wystąpił niskopoziomowy błąd operacji wejścia i wyjścia (jest to alarmujący komunikat).

`EISDIR`
: Końcowy element ścieżki `path` lub interpretera jest katalogiem.

`ELOOP`
: System napotkał zbyt wiele dowiązań symbolicznych podczas analizowania ścieżki.

`EMFILE`
: Proces wywołujący osiągnął ograniczenie liczby otwartych plików.

`ENFILE`
: Zostało osiągnięte systemowe ograniczenie dotyczące liczby otwartych plików.

`ENOENT`
: Docelowa ścieżka `path`, plik `file` lub wymagana biblioteka współdzielona nie istnieją.

`ENOEXEC`
: Docelowa ścieżka `path` lub plik `file` jest błędnym obrazem binarnym bądź przystosowanym do działania w innej architekturze maszynowej.

`ENOMEM`
: Nie istnieje wystarczająco dużo pamięci jądra, aby uruchomić nowy program.

`ENOTDIR`
: Pośredni element w ścieżce `path` nie jest katalogiem.

`EPERM`
: System plików, w którym znajduje się ścieżka `path` lub plik `file`, jest zamontowany w trybie `nosuid`. Użytkownik nie ma uprawnień administratora, a ścieżka `path` lub plik `file` mają ustawione bity `suid` lub `sgid`.

`ETXTBSY`
: Docelowa ścieżka `path` lub plik `file` jest otwarty do zapisu przez inny proces.

Funkcja systemowa fork()

Nowy proces, działający z takim samym obrazem binarnym jak aktualny, może zostać utworzony poprzez wywołanie funkcji systemowej `fork()`:

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork (void);
```

Poprawne wywołanie funkcji `fork()` tworzy nowy proces, niemal identyczny z procesem wywołującym. Po zakończeniu funkcji `fork()`, jeśli nic specjalnego się nie wydarzy, będą jednocześnie działać oba procesy.

Nowy proces zwany jest „potomkiem" procesu pierwotnego, a ten z kolei zwany jest „rodzicem". Dla potomka, poprawne wywołanie funkcji `fork()` zwraca zero. Dla rodzica, funkcja `fork()` zwraca PID potomka. Procesy potomka i rodzica są prawie identyczne, za wyjątkiem kilku niezbędnych różnic:

- Potomkowi zostaje przydzielony nowy identyfikator procesu; oczywiście różni się on od PID rodzica.
- Identyfikator procesu rodzicielskiego dla potomka ustawiony zostaje na PID jego rodzica.
- Statystyki zasobów dla potomka zostają wyzerowane.
- Wszystkie oczekujące sygnały zostają usunięte i nie będą dziedziczone przez potomka (szczegóły: rozdział 9.).
- Żadne uzyskane blokady plików nie będą dziedziczone przez potomka.

W przypadku błędu proces potomka nie jest tworzony, funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno`. Istnieją dwie możliwe wartości zmiennej `errno`, posiadające trzy znaczenia:

`EAGAIN`
: Jądro napotkało problem podczas przydzielania pewnych zasobów (np. nowego identyfikatora procesu) lub zostało osiągnięte ograniczenie zasobów (*rlimit*) o nazwie `RLIMIT_NPROC` (opis: rozdział 6.).

`ENOMEM`
: Nie istnieje wystarczająca ilość pamięci jądra, aby zakończyć tę operację.

Użycie funkcji jest proste:

```
pid_t pid;

pid = fork ( );
if (pid > 0)
   printf ("Jestem rodzicem procesu PID=%d!\n", pid);
else if (!pid)
   printf ("Jestem potomkiem!\n");
else if (pid == -1)
   perror ("fork");
```

Najbardziej powszechnym użyciem funkcji `fork()` jest utworzenie kolejnego procesu, dla którego następuje załadowanie nowego obrazu binarnego — może to być na przykład powłoka systemowa, która uruchamia nowy program dla użytkownika lub proces wywołujący program pomocniczy. Proces taki najpierw rozwidla się, a następnie jego potomek uruchamia nowy obraz

binarny. Ten zestaw działań „rozwidlenie i uruchomienie" jest prostą i częstą czynnością w systemie. Następujący przykład prezentuje tworzenie nowego procesu, uruchamiającego obraz binarny */bin/windlass*:

```
pid_t pid;

pid = fork ( );
if (pid == -1)
   perror ("fork");

/* potomek ... */
if (!pid)
{
   const char *args[] = { "windlass", NULL };
   int ret;

   ret = execv ("/bin/windlass", args);
   if (ret == -1)
   {
      perror ("execv");
      exit (EXIT_FAILURE);
   }
}
```

Po wykonaniu funkcji `fork()` proces rodzicielski działa w dalszym ciągu bez zmian, w przeciwieństwie do swojego potomka. Wywołanie funkcji `execv()` nakazuje potomkowi uruchomienie programu */bin/windlass*.

Kopiowanie podczas zapisu

We wczesnych wersjach systemu Unix operacja rozwidlenia była prosta, a nawet prymitywna. Po wywołaniu jądro tworzyło kopię wszystkich wewnętrznych struktur danych, powielało pozycje tablicy stron dla procesu, a następnie wykonywało kopiowanie strona po stronie z przestrzeni adresowej rodzica do nowej przestrzeni adresowej potomka. Jednak kopiowanie strona po stronie było (przynajmniej dla jądra) operacją, która zajmowała dużo czasu.

Nowoczesne systemy uniksowe (np. Linux) zachowują się bardziej optymalnie. Zamiast wykonywać kopiowanie całej przestrzeni adresowej, używają stron z zaimplementowanym mechanizmem *kopiowania podczas zapisu* (ang. *copy-on-write*, w skrócie *COW*).

Kopiowanie podczas zapisu jest strategią leniwej optymalizacji, zaprojektowaną, aby łagodzić obciążenie pojawiające się podczas kopiowania zasobów. Założenie jest proste: jeśli wielu konsumentów wymaga dostępu z uprawnieniami odczytu do takich samych zasobów, wówczas powielanie zasobów nie jest wymagane. Zamiast tego każdy konsument może otrzymać wskaźnik do tego samego zasobu. Dopóki żaden z konsumentów nie będzie próbował zmodyfikować swojej „kopii" zasobów, dopóty będzie istnieć wrażenie, że ma do niej wyłączny dostęp, a problem obciążenia podczas kopiowania nie będzie obecny. Jeśli konsument faktycznie spróbuje zmodyfikować swoje zasoby, wówczas w sposób dla niego niezauważalny nastąpi ich powielenie, a następnie zostanie mu przydzielona nowo utworzona kopia zasobów. Konsument może potem modyfikować swoją własną kopię zasobów, podczas gdy inni w dalszym ciągu będą współdzielić oryginalną, niezmienioną wersję. Stąd nazwa: *kopiowanie* występuje tylko *podczas zapisu*.

Podstawową korzyścią zastosowania tego rozwiązania jest pewność, że jeśli konsument nigdy nie zmodyfikuje swoich zasobów, wykonanie kopii nigdy nie będzie potrzebne. Zastosowanie ma tu także ogólna zaleta algorytmów leniwych, czyli czekanie z wykonaniem kosztownych działań do ostatniego momentu.

W tym szczególnym przypadku pamięci wirtualnej, kopiowanie podczas zapisu jest zaimplementowane dla każdej strony z osobna. Dlatego też, dopóki proces nie modyfikuje swojej całej przestrzeni adresowej, jej pełna kopia nie jest wymagana. Po zakończeniu operacji rozwidlenia, rodzic i potomek otrzymują informację o przydzieleniu im unikalnych przestrzeni adresowych, gdy w rzeczywistości współdzielą oryginalne strony procesu potomnego, które z kolei mogą być współdzielone z innymi procesami rodziców i potomków itd.!

Implementacja dla jądra wykonana jest w prosty sposób. W wewnętrznych strukturach jądra, dotyczących stronicowania, strony z zasobami posiadają specjalne atrybuty, informujące, iż są one tylko do odczytu oraz że mają uaktywniony mechanizm kopiowania podczas zapisu. Jeśli inny proces spróbuje zmodyfikować taką stronę, spowoduje to wygenerowanie dla niej błędu. Jądro obsłuży następnie ten błąd poprzez wykonanie niezauważalnej dla procesu kopii strony. W tym momencie istniejący do tej pory atrybut strony, informujący o aktywnej dla niej opcji kopiowania podczas zapisu, zostanie usunięty, a strona nie będzie już dłużej współdzielona.

Ponieważ współczesne architektury maszynowe udostępniają wsparcie na poziomie sprzętowym dla kopiowania podczas zapisu w swoich układach zarządzania pamięcią (MMU), ten algorytm jest prosty i łatwy do zaimplementowania.

Jeszcze większą korzyść z zastosowania mechanizmu kopiowania podczas zapisu osiąga się w przypadku rozwidlania procesów. Ponieważ w większości przypadków rozwidlenia połączone są z występującymi zaraz po nich operacjami wywołania funkcji *exec*, dlatego też kopiowanie przestrzeni adresowej rodzica do przestrzeni adresowej potomka jest stratą czasu: jeśli proces potomny w końcu uruchomi nowy obraz binarny, poprzednia zawartość jego przestrzeni adresowej zostanie usunięta. Kopiowanie podczas zapisu optymalizuje takie przypadki.

Funkcja vfork()

Zanim pojawiły się strony z mechanizmem kopiowania podczas zapisu, projektanci Uniksa zajmowali się rozrzutną operacją kopiowania przestrzeni adresowej podczas wykonywania rozwidlenia procesów, po której następowało uruchomienie obrazu binarnego. Projektanci systemu BSD stworzyli w wersji 3.0 tego systemu funkcję systemową `vfork()`:

```
#include <sys/types.h>
#include <unistd.h>

pid_t vfork (void);
```

Poprawnie wywołana funkcja `vfork()` wykonuje te same działania co funkcja `fork()`, za wyjątkiem tego, że proces potomny musi natychmiast przeprowadzić udane wywołanie jednej funkcji z grupy *exec* lub wyjść za pomocą funkcji `_exit()` (omówionej w następnym podrozdziale). Funkcja systemowa `vfork()` chroni przestrzeń adresową oraz kopie tablicy stron poprzez zawieszenie procesu rodzica do czasu, gdy proces potomny zakończy swoje działanie lub uruchomi nowy obraz binarny. W międzyczasie rodzic i potomek współdzielą (bez użycia semantyki kopiowania podczas zapisu) swoją przestrzeń adresową oraz wpisy w tablicy stron. W rzeczywistości jedynym działaniem, jakie wykonuje funkcja `vfork()`, jest powielenie wewnętrznych struktur danych dla jądra. Zgodnie z tym proces potomny nie może modyfikować żadnego fragmentu pamięci w przestrzeni adresowej.

Funkcja systemowa `vfork()` jest reliktem i nie powinna nigdy zostać zaimplementowana w Linuksie, choć należy zwrócić uwagę, że nawet uwzględniając mechanizm kopiowania podczas zapisu, `vfork()` jest wciąż szybsza od funkcji `fork()`, ponieważ wpisy w tablicy

stron nie muszą być kopiowane[2]. Mimo to po pojawieniu się stron używających mechanizmu kopiowania podczas zapisu, wszystkie argumenty skierowane przeciwko wykorzystaniu funkcji `fork()` tracą na wartości. W rzeczywistości, do wersji 2.2.0 jądra Linuksa, funkcja `vfork()` była zwykłym interfejsem programowym do funkcji `fork()`.Taka implementacja była możliwa, ponieważ wymagania dotyczące funkcji `vfork()` były mniejsze od wymagań dla funkcji `fork()`.

Żadna z implementacji funkcji `vfork()` nie jest pozbawiona błędów. Przeanalizujmy sytuację, gdy wywołanie funkcji *exec* nie powiedzie się! Rodzic będzie wówczas zawieszony do momentu, gdy potomek zakończy swoją pracę; w przeciwnym przypadku będzie on cały czas oczekiwać, podczas gdy proces potomny „zastanawia się", co powinien uczynić.

Zakończenie procesu

POSIX oraz C98 definiują standardową funkcję pozwalającą zakończyć aktualny proces:

```
#include <stdlib.h>
void exit (int status);
```

Funkcja `exit()` przeprowadza pewne podstawowe operacje związane z zamykaniem procesu, a następnie nakazuje jądru zakończenie jego działania. Funkcja nie może zwrócić błędu — w rzeczywistości nigdy nie wróci do programu, z którego została wywołana[3]. Dlatego też nie ma sensu umieszczanie jakichkolwiek instrukcji po wywołaniu funkcji `exit()`.

Parametr `status` jest używany, aby określić stan zakończenia procesu. Inne programy — jak również użytkownik powłoki systemowej — mogą sprawdzać tę wartość. Dokładnie rzecz ujmując, do procesu rodzicielskiego zwrócona zostaje wartość `status & 0377`. Odzyskiwanie wartości powrotnej zostanie omówione w dalszej części tego rozdziału.

Stałe `EXIT_SUCCESS` oraz `EXIT_FAILURE` są przenośnymi rozwiązaniami pozwalającymi reprezentować stan poprawnego oraz błędnego wykonania. W Linuksie wartość `0` zwykle oznacza sukces; wartości niezerowe, takie jak `1` lub `−1`, odpowiadają błędom.

Dlatego też wyjście z informacją o poprawnym wykonaniu będzie tak proste, jak wskazuje na to poniższy kod:

```
exit(EXIT_SUCCESS);
```

Zanim proces zostanie zakończony, biblioteka języka C przeprowadza następujące operacje związane z jego zamykaniem:

1. Nastąpi wykonanie wszystkich funkcji zarejestrowanych przy użyciu `atexit()` lub `on_exit()`, w odwrotnej kolejności do ich rejestrowania (funkcje te zostaną omówione w dalszej części rozdziału).
2. Zostaną opróżnione wszystkie otwarte strumienie dla standardowych operacji wejścia i wyjścia (szczegóły w rozdziale 3.).
3. Zostaną usunięte wszystkie tymczasowe pliki utworzone za pomocą funkcji `tmpfile()`.

[2] Poprawka, implementująca mechanizm kopiowania podczas zapisu dla współdzielonych elementów tablicy stron, pojawiała się na liście dyskusyjnej jądra Linuksa, lecz do tej pory nie stała się częścią jądra w wersji 2.6. Jeśli zostanie dołączona do jądra, spowoduje to, że nie będą istnieć żadne korzyści z używania funkcji `vfork()`.

[3] Zakończy go — *przyp. tłum.*

Wykonanie powyższych czynności kończy pracę w przestrzeni użytkownika, dlatego też funkcja `exit()` może obecnie wywołać funkcję systemową `_exit()`, aby pozwolić jądru na obsługę pozostałej części operacji usuwania procesu:

```
#include <unistd.h>

void _exit (int status);
```

Gdy proces zostaje usunięty, jądro likwiduje zasoby, które wcześniej zostały stworzone dla tego procesu i nie będą już więcej używane. Należą do nich między innymi takie zasoby jak: przydzielona pamięć, otwarte pliki oraz semafory typu System V. Po wykonaniu porządkowania jądro usuwa proces oraz powiadamia rodzica o zlikwidowaniu potomka.

Aplikacje mogą również bezpośrednio wywoływać funkcję `_exit()`, lecz taka operacja rzadko ma sens: większość programów wymaga wykonania pewnego porządkowania (na przykład opróżnienia strumienia *stdout*), które udostępnia pełna funkcja `exit()`. Należy zauważyć jednak, że użytkownicy funkcji `vfork()` powinni po wykonaniu rozwidlenia używać wywołania funkcji `_exit()` zamiast `exit()`.

Olśniewającym przykładem nadmiarowości jest funkcja `_Exit()`, która zdefiniowana została przez standard ISO C99 i posiada zachowanie identyczne z funkcją `_exit()`:

```
#include <stdlib.h>
void _Exit (int status);
```

Inne sposoby na zakończenie procesu

Klasycznym sposobem, pozwalającym na zakończenie działania programu, nie jest jawne wywołanie funkcji systemowej, lecz dojście do końca programu. W przypadku języka C występuje to wówczas, gdy kończy się wykonywanie funkcji `main()`. Ten sposób mimo wszystko wymaga jednak wykonania funkcji systemowej: kompilator po prostu wstawia domyślne wywołanie funkcji `_exit()` po swoim kodzie zamykającym aplikację. Dobrą praktyką programistyczną jest jawne zwracanie kodu powrotu przy użyciu funkcji `exit()` lub bezpośrednim zwróceniu wartości z funkcji `main()`. Powłoka systemowa używa kodu powrotu, aby ocenić, czy wykonanie polecenia zakończyło się sukcesem lub niepowodzeniem. Należy zwrócić uwagę na to, że zakończenie sukcesem oznacza wywołanie funkcji `exit(O)` lub wyjście z funkcji `main()` z wartością równą zeru.

Proces może również zostać zakończony, jeśli otrzyma sygnał, którego domyślnym działaniem będzie przerwanie tego procesu. Takimi sygnałami są między innymi `SIGTERM` i `SIGKILL` (omówione w rozdziale 9.).

Wreszcie, ostatnim sposobem na zakończenie procesu jest „rozgniewanie" jądra. Jądro może usunąć proces, który wykonał błędną instrukcję, spowodował naruszenie segmentacji, wykorzystał całą pamięć itp.

Funkcja atexit()

POSIX 1003.1-2001 definiuje, a Linux implementuje funkcję biblioteczną `atexit()` używaną w celu rejestrowania funkcji, które mają zostać wywołane podczas operacji usuwania procesu:

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

Poprawne wywołanie funkcji `atexit()` rejestruje daną funkcję w celu jej automatycznego uruchomienia podczas normalnego kończenia działania procesu, to znaczy, gdy proces jest usuwany poprzez wywołanie funkcji `exit()` lub powrót z funkcji `main()`. Jeśli proces wywołuje funkcję z grupy *exec*, wówczas lista zarejestrowanych funkcji zostaje zlikwidowana (funkcje nie istnieją już w przestrzeni adresowej nowego procesu). Jeśli proces zostanie zakończony przez sygnał, wówczas zarejestrowane funkcje nie będą wywołane.

Funkcja, która ma zostać zarejestrowana, nie posiada parametrów ani też nie zwraca żadnej wartości. Prototyp funkcji wygląda następująco:

```
void my_function (void);
```

Funkcje zostają wywołane w odwrotnej kolejności do ich zarejestrowania. Oznacza to, że funkcje przechowywane są na stosie i ostatnia zapamiętana będzie uruchomiona jako pierwsza (stos LIFO). Funkcje zarejestrowane nie mogą wywoływać funkcji `exit()`, gdyż spowoduje to powstanie nieskończonego cyklu rekurencji. Jeśli funkcja musi wcześniej zakończyć operację usuwania procesu, powinna wówczas wywołać `_exit()`. Jednak takie zachowanie nie jest zalecane, ponieważ któraś z ważnych funkcji może wtedy nie zostać uruchomiona.

Standard POSIX wymaga, aby funkcja `atexit()` pozwalała na zarejestrowanie tylu funkcji, ile wynosi stała `ATEXIT_MAX`. Wartość ta musi być równa co najmniej 32. Dokładna wartość może być otrzymana za pomocą wywołania funkcji `sysconf()` z parametrem `_SC_ATEXIT_MAX`:

```
long atexit_max;

atexit_max = sysconf (_SC_ATEXIT_MAX);
printf ("atexit_max=%ld\n", atexit_max);
```

W przypadku sukcesu funkcja zwraca `0`, a w przypadku błędu `-1`.

Oto prosty przykład:

```
#include <stdio.h>
#include <stdlib.h>

void out (void)
{
   printf ("Wywołanie funkcji atexit( ) wykonało się poprawnie!\n");
}

int main (void)
{
   if (atexit (out))
      fprintf(stderr, "Wywołanie funkcji atexit( ) nie powiodło się!\n");

   return 0;
}
```

Funkcja on_exit()

W systemie operacyjnym SunOS 4 zdefiniowano własny równoważnik funkcji `atexit()`, który posiada wsparcie w bibliotece Linuksa *glibc*:

```
#include <stdlib.h>

int on_exit (void (*function)(int , void *), void *arg);
```

Funkcja ta działa tak samo jak funkcja `atexit()`, lecz prototyp dla rejestrowanej funkcji jest inny:

```
void my_function (int status, void *arg);
```

Parametr `status` jest wartością przekazywaną do funkcji `exit()` lub zwracaną przy wyjściu z funkcji `main()`. Parametr `arg` jest drugim argumentem przekazywanym do funkcji `on_exit()`. W momencie wywołania funkcji należy zadbać o to, aby obszar pamięci wskazywany przez `arg` był poprawny.

Najnowsza wersja systemu Solaris nie wspiera już tej funkcji. Zamiast niej należy używać funkcji `atexit()`, która jest zgodna ze standardem.

Sygnał SIGCHLD

Podczas operacji zakończenia procesu jądro wysyła do rodzica sygnał `SIGCHLD`. Domyślnie jest on ignorowany i proces rodzicielski nie podejmuje żadnych działań. Proces może jednak obsłużyć ten sygnał, używając funkcji systemowych `signal()` lub `sigaction()`. Funkcje te, a także cała reszta bogatego świata sygnałów, opisane zostaną w rozdziale 9.

Sygnał `SIGCHLD` może zostać wygenerowany lub wysłany w dowolnym momencie, ponieważ kończenie procesu potomka działa w sposób asynchroniczny w stosunku do jego procesu rodzicielskiego. Często jednak rodzic chce wiedzieć coś więcej na temat kończenia procesu potomka lub nawet jawnie czeka na to zdarzenie. Jest to możliwe przy użyciu funkcji systemowych, które zostaną wkrótce przeanalizowane.

Oczekiwanie na zakończone procesy potomka

Otrzymywanie potwierdzenia poprzez odbiór sygnału jest oczywiście pożytecznym działaniem, lecz wiele procesów rodzicielskich chce uzyskać więcej informacji, gdy jeden z ich procesów potomnych jest kończony — na przykład kod powrotu potomka.

Gdyby proces potomny całkowicie znikał podczas jego kończenia, wówczas — jak można się tego spodziewać — nie pozostawałoby nic, co proces rodzicielski mógłby badać. Dlatego też początkowi projektanci systemu Unix zdecydowali, że w momencie, gdy proces potomny jest usuwany przed procesem rodzicielskim, jądro powinno przypisać potomkowi specjalny stan procesu. Proces w takim stanie zwany jest *procesem zombie*. Pozostaje tylko minimalny szkielet po tym, czym kiedyś był proces — pewne podstawowe struktury danych w jądrze, które zawierają potencjalnie użyteczne informacje. Proces w tym stanie czeka na swojego rodzica, aby wysłał zapytanie dotyczące jego statusu (jest to procedura zwana *obsługiwaniem* procesu zombie). Dopiero po tym, gdy rodzic otrzymuje informacje dotyczące zakończonego procesu potomnego, następuje formalne usunięcie potomka i przestaje on istnieć nawet w stanie zombie.

Jądro Linuksa dostarcza paru interfejsów w celu uzyskiwania informacji o zakończonych potomkach. Najprostszym z nich, zdefiniowanym przez POSIX, jest funkcja `wait()`:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
```

Wywołanie funkcji `wait()` zwraca identyfikator procesu zakończonego potomka lub `-1` w przypadku błędu. Jeśli żaden proces potomny nie jest jeszcze zakończony, funkcja blokuje się, dopóki

potomek nie zostanie zlikwidowany. Jeśli proces potomny jest już zakończony, funkcja natychmiast kończy swoje działanie. Dlatego też wywołanie funkcji `wait()` w odpowiedzi na posiadaną informację o tym, że proces potomny jest zakończony — na przykład, po otrzymaniu sygnału `SIGCHLD` — finalizuje się zawsze bez blokowania.

W przypadku błędu istnieją dwie możliwe wartości zmiennej `errno`:

`ECHILD`
: Proces wywołujący nie posiada żadnych potomków.

`EINTR`
: Podczas oczekiwania otrzymano sygnał i funkcja zakończyła się przed czasem.

Jeśli wskaźnik `status` nie jest równy `NULL`, zawiera on na dodatkowe informacje o procesie potomnym. Ponieważ POSIX zezwala, aby implementacje w dowolny sposób definiowały bity w parametrze `status`, dlatego też standard dostarcza zestawu makr, pozwalających odpowiednio interpretować te informacje:

```
#include <sys/wait.h>

int WIFEXITED (status);
int WIFSIGNALED (status);
int WIFSTOPPED (status);
int WIFCONTINUED (status);

int WEXITSTATUS (status);
int WTERMSIG (status);
int WSTOPSIG (status);
int WCOREDUMP (status);
```

W zależności od tego, jak został zakończony proces, jedno z dwóch pierwszych makr może zwrócić wartość prawdy (wartość niezerową). Pierwsze makro, `WIFEXITED`, zwraca wartość prawdy, jeśli proces został zakończony w normalny sposób, to znaczy, jeśli została wywołana funkcja `_exit()`. W tym przypadku makro `WEXITSTATUS` zwraca osiem mniej znaczących bitów z wartości, które zostały przekazane do funkcji `_exit()`.

Makro `WIFSIGNALED` zwraca wartość prawdy, jeśli zakończenie procesu zostało spowodowane przez sygnał (w rozdziale 9. przeprowadzona zostanie analiza sygnałów). W tym przypadku `WTERMSIG` zwraca numer sygnału, który spowodował zakończenie procesu, natomiast `WCOREDUMP` zwraca wartość prawdy, jeśli proces wykonał zrzut systemowy w odpowiedzi na otrzymanie sygnału. Makro `WCOREDUMP` nie jest zdefiniowane w standardzie POSIX, choć wspiera go wiele systemów uniksowych (w tym Linux).

Makra `WIFSTOPPED` oraz `WIFCONTINUED` zwracają wartość prawdy, jeśli proces został odpowiednio zatrzymany lub kontynuuje swoje działanie, a obecnie jest śledzony za pomocą funkcji systemowej `ptrace()`. Wykorzystanie tych makr ma zazwyczaj sens wówczas, gdy używa się debuggera, chociaż w przypadku użycia funkcji `waitpid()` (opisanej w następnym podrozdziale) mogą one być również zastosowane, by zaimplementować zarządzanie zadaniami. Zazwyczaj tylko funkcja `wait()` używana jest w celu przekazania informacji o zakończeniu procesu. Jeśli makro `WIFSTOPPED` zwróci wartość prawdy, wówczas `WSTOPSIG` dostarczy informacji o numerze sygnału, który spowodował zakończenie procesu. Makro `WIFCONTINUED` nie jest zdefiniowane w standardzie POSIX, choć następne standardy definiują je dla funkcji `waitpid()`. W przypadku wersji 2.6.10 jądra Linuksa istnieje również definicja tego makra dla funkcji `wait()`.

Oto przykład programu, który używa funkcji `wait()` w celu ustalenia, co przydarzyło się procesowi potomnemu:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main (void)
{
   int status;
   pid_t pid;

   if (!fork ( ))
      return 1;

   pid = wait (&status);
   if (pid == -1)
      perror ("wait");

   printf ("pid=%d\n", pid);

   if (WIFEXITED (status))
      printf ("Zwykłe zakończenie procesu z kodem powrotu=%d\n", WEXITSTATUS (status));

   if (WIFSIGNALED (status))
      printf ("Zniszczony przez sygnał=%d%s\n", WTERMSIG (status), WCOREDUMP (status) ?
        " (wykonano zrzut systemowy)" : "");

   if (WIFSTOPPED (status))
      printf ("Zatrzymany przez sygnał=%d\n", WSTOPSIG (status));

   if (WIFCONTINUED (status))
      printf ("Kontynuuje swoje działanie\n");

   return 0;
}
```

Program ten tworzy proces potomny, który natychmiast kończy swoje działanie. Proces rodzicielski wywołuje następnie funkcję systemową `wait()`, aby ustalić status potomka. Proces wypisuje identyfikator potomka, a także informacje, w jaki sposób zakończył on swoje działanie. Ponieważ dla tego przypadku proces potomka zakończony zostanie przez powrót z funkcji `main()`, można przypuszczać, że na wyjściu pojawi się następujący wynik:

```
$ ./wait
pid=8529
Zwykłe zakończenie procesu z kodem powrotu=1
```

Jeśli jednak zamiast normalnego zakończenia procesu potomka wywołana zostanie przez niego funkcja `abort()`[4], która wyśle mu sygnał `SIGABRT`, wówczas na wyjściu pojawi się inny wynik:

```
$ ./wait
pid=8678
Zniszczony przez sygnał=6
```

[4] Zdefiniowana w pliku nagłówkowym `<stdlib.h>`.

Oczekiwanie na określony proces

Obserwowanie zachowania procesów potomnych ma duże znaczenie. Często jednak proces posiada wiele potomków i nie chce oczekiwać na wszystkich, lecz tylko na pewien określony proces potomny. Jednym z rozwiązań mogłoby być wielokrotne wywołanie funkcji `wait()` i za każdym razem sprawdzanie jej kodu powrotu. Jest to jednak niewygodne — co będzie w przypadku, gdy programista zechce później sprawdzić status innego zakończonego procesu? Rodzic musiałby zapisywać wszystkie wyniki wywołań funkcji `wait()`, aby ewentualnie mógł się do nich później odwołać.

Jeśli znany jest PID dla procesu, który należy obserwować, wówczas można użyć funkcji systemowej `waitpid()`:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid (pid_t pid, int *status, int options);
```

Funkcja `waitpid()` posiada więcej możliwości niż funkcja `wait()`. Jej dodatkowe parametry pozwalają na precyzyjne dostrojenie działania.

Parametr `pid` określa, jakiego procesu lub procesów należy oczekiwać. Wartości tego parametru zawierają się w czterech zakresach liczbowych:

`< -1`
: Należy oczekiwać na dowolny proces potomny, dla którego identyfikator grupy procesów równy jest wartości bezwzględnej parametru `pid`. Na przykład, przekazanie wartości `-500` spowoduje oczekiwanie na dowolny proces z grupy 500.

`-1`
: Należy oczekiwać na dowolny proces potomny. Jest to zachowanie identyczne z zachowaniem funkcji `wait()`.

`0`
: Należy oczekiwać na dowolny proces potomny, który należy do tej samej grupy procesów co proces wywołujący.

`> 0`
: Należy oczekiwać na proces potomny, którego identyfikator równy jest dokładnie wartości przekazanej w parametrze `pid`. Na przykład, przekazanie wartości `500` spowoduje oczekiwanie na proces z PID równym 500.

Parametr `status` działa tak samo jak jedyny parametr funkcji `wait()` i może zostać użyty w poprzednio omówionych makrach.

Parametr `options` jest zerem lub sumą binarną jednego lub więcej poniższych znaczników:

`WNOHANG`
: Funkcja nie powinna się zablokować, lecz natychmiast wrócić, jeśli żaden z pasujących procesów potomnych nie został zakończony (lub zatrzymany albo ponownie uruchomiony).

`WUNTRACED`
: Jeśli ten znacznik zostanie użyty, wówczas makro `WIFSTOPPED` zwróci wartość prawdy, nawet jeśli proces wywołujący nie śledzi procesu potomnego. Ten znacznik pozwala na zaimplementowanie bardziej ogólnego sterowania zadaniami, podobnie jak ma to miejsce w powłoce systemowej.

`WCONTINUED`
: Jeśli ten znacznik zostanie użyty, wówczas makro `WIFCONTINUED` zwróci wartość prawdy, nawet jeśli proces wywołujący nie śledzi procesu potomnego. Podobnie jak w przypadku znacznika `WCONTINUED` jest on przydatny podczas implementacji powłoki systemowej.

W przypadku sukcesu funkcja `waitpid()` zwraca identyfikator procesu, którego status zmienił się. Jeśli użyto znacznika `WNOHANG`, a określony proces lub procesy potomne nie zmieniły jeszcze swojego statusu, funkcja zwraca `0`. W przypadku błędu, funkcja zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych trzech wartości:

`ECHILD`
: Proces lub procesy określone w parametrze `pid` nie istnieją lub nie są potomkami procesu wywołującego.

`EINTR`
: Opcja `WNOHANG` nie została użyta, ale podczas oczekiwania odebrano sygnał.

`EINVAL`
: Parametr `options` jest niepoprawny.

Jako przykład przedstawiony zostanie program, który pobiera kod powrotu określonego potomka o identyfikatorze PID równym 1742, lecz wraca natychmiast, gdy proces tego potomka nie został jeszcze zakończony. Rozwiązanie tego problemu może być podobne do poniższego kodu:

```
int status;
pid_t pid;

pid = waitpid (1742, &status, WNOHANG);
if (pid == -1)
   perror ("waitpid");
else
{
   printf ("pid=%d\n", pid);

   if (WIFEXITED (status))
      printf ("Zwykłe zakończenie procesu z kodem powrotu=%d\n", WEXITSTATUS (status));

   if (WIFSIGNALED (status))
      printf ("Zniszczony przez sygnał=%d%s\n", WTERMSIG (status), WCOREDUMP (status) ?
        " (wykonano zrzut systemowy)" : "");
}
```

Na koniec warto zwrócić uwagę na to, że następujący sposób użycia funkcji `wait()`:

```
wait (&status);
```

jest identyczny z poniższym użyciem funkcji `waitpid()`:

```
waitpid (-1, &status, 0);
```

Jeszcze wszechstronniejsza funkcja oczekiwania

W rozszerzeniu XSI dla standardu POSIX zdefiniowano, a w systemie Linux zaimplementowano funkcję `waitid()`, która może zostać zastosowana w aplikacjach wymagających jeszcze większej uniwersalności dla zadań wymagających oczekiwania na potomków:

```
#include <sys/wait.h>

int waitid (idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Podobnie jak funkcje `wait()` i `waitpid()` funkcja `waitid()` używana jest, aby oczekiwać na proces potomny i uzyskiwać informację dotyczącą zmian jego statusu (zakończenia, zatrzymania, kontynuowania pracy). Dostarcza ona nawet większej liczby opcji niż jej poprzedniczki, co jest związane z jej zwiększoną złożonością.

Podobnie jak w przypadku `waitpid()` funkcja `waitid()` pozwala projektantowi oprogramowania określać, na co należy oczekiwać. Jednakże funkcja `waitid()` nie używa do tego celu jednego, lecz dwóch parametrów. Parametry `idtype` oraz `id` określają, na jakie procesy potomne należy czekać i realizują ten sam cel, któremu służyło użycie pojedynczego parametru `pid` w funkcji `waitpid()`. Parametr `idtype` może być równy jednej z poniższych wartości:

`P_PID`
: Należy oczekiwać na potomka, dla którego identyfikator równy jest parametrowi `id`.

`P_GID`
: Należy oczekiwać na potomka, dla którego identyfikator grupy procesów równy jest parametrowi `id`.

`P_ALL`
: Należy oczekiwać na dowolny proces potomny; parametr `id` jest ignorowany.

Parametr `id` posiada rzadko używany typ `id_t`, który reprezentuje ogólny numer identyfikacyjny. Zastosowano go, aby zapewnić kompatybilność w przypadku, gdy w następnych implementacjach wprowadzona zostanie nowa wartość `idtype` i przypuszczalnie uzyska się dzięki temu lepsze zabezpieczenie zachowania nowo utworzonego identyfikatora. Typ ten gwarantuje wystarczający zakres danych, aby przechowywać dowolną wartość typu `pid_t`. W przypadku Linuksa projektanci mogą używać nowego typu w taki sam sposób jak w przypadku typu `pid_t` — na przykład, przez bezpośrednie przekazywanie wartości lub stałych liczbowych o typie `pid_t`. Skrupulatni programiści mogą jednak bez problemu rzutować typy.

Parametr `options` jest sumą bitową jednego lub więcej następujących znaczników:

`WEXITED`
: Funkcja będzie oczekiwać na procesy potomne (określone przez `id` i `idtype`), które zostały zakończone.

`WSTOPPED`
: Funkcja będzie oczekiwać na procesy potomne, które wstrzymały swoje wykonywanie w odpowiedzi na odebrany sygnał.

`WCONTINUED`
: Funkcja będzie oczekiwać na procesy potomne, które kontynuują swoje działanie w odpowiedzi na odebrany sygnał.

`WNOHANG`
: Funkcja nigdy się nie zablokuje, lecz natychmiast wróci, jeśli odpowiedni proces potomny już się zakończył (lub zatrzymał albo kontynuuje swoje działanie).

`WNOWAIT`
: Funkcja nie usunie odpowiedniego procesu z grupy procesów zombie. Proces może zostać obsłużony w przyszłości.

Po udanym obsłużeniu procesu potomnego funkcja `waitid()` uaktualnia parametr `infop`, który musi wskazywać na poprawną zmienną o typie `siginfo_t`. Dokładna definicja struktury

siginfo_t zależy od implementacji[5], lecz po wywołaniu funkcji waitid() kilka jej pól jest poprawnych. Oznacza to, że udane wywołanie funkcji zapewni, iż następujące pola zostaną wypełnione:

si_pid
: PID potomka.

si_uid
: UID potomka.

si_code
: W odpowiedzi na zakończenie procesu potomnego lub na zmiany statusu wywołane otrzymaniem sygnału, takie jak usunięcie procesu, zatrzymanie działania lub kontynuowanie pracy, pole to może zostać odpowiednio ustawione na jedną z wartości CLD_EXITED, CLD_KILLED, CLD_STOPPED lub CLD_CONTINUED.

si_signo
: Pole ustawione zostaje na SIGCHLD.

si_status
: Jeśli pole si_code równe jest CLD_EXITED, wówczas pole si_status równe jest kodowi powrotu procesu potomnego. W przeciwnym razie pole to jest numerem sygnału, dostarczonego do potomka, który spowodował zmianę statusu.

W przypadku sukcesu, funkcja waitid() zwraca wartość zero. W przypadku błędu, zwraca −1 oraz odpowiednio ustawia zmienną errno na jedną z poniższych wartości:

ECHLD
: Proces lub procesy określone w parametrach id i idtype nie istnieją.

EINTR
: Znacznik WNOHANG nie został ustawiony w opcjach, ale pojawił się sygnał, który przerwał wykonywanie funkcji.

EINVAL
: Parametr options lub połączenie argumentów id i idtype są niepoprawne.

Funkcja waitid() dostarcza dodatkowej, użytecznej semantyki, która nie istnieje w przypadku funkcji wait() i waitpid(). W szczególności użyteczna może okazać się informacja uzyskana ze struktury siginfo_t. Jeśli jednak taka informacja nie jest wymagana, być może ma sens użycie prostszych funkcji, które posiadają wsparcie w wielu systemach i mają szansę na implementację w większej liczbie systemów nielinuksowych.

BSD wkracza do akcji: funkcje wait3() i wait4()

Funkcja waitpid() została stworzona dla systemu operacyjnego AT&T System V Release 4, natomiast w systemie BSD zdecydowano się na własne rozwiązanie i dostarczono dwie inne funkcje używane w celu oczekiwania na zmianę statusu potomka:

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
```

[5] Struktura siginfo_t w systemie Linux jest bardzo skomplikowana. Jej definicja zapisana jest w pliku */usr/include/bits/siginfo.h*. Struktura ta zostanie dokładniej przeanalizowana w rozdziale 9.

```
#include <sys/wait.h>

pid_t wait3 (int *status, int options, struct rusage *rusage);
pid_t wait4 (pid_t pid, int *status, int options, struct rusage *rusage);
```

Liczby 3 i 4, pojawiające się w ich nazwach, wskazują na to, że są one odpowiednikami funkcji `wait()`, używającymi trzech i czterech parametrów.

Funkcje te działają podobnie do funkcji `waitpid()`, ale używają dodatkowego parametru `rusage`. Następujące wywołanie funkcji `wait3()`:

```
pid = wait3 (status, options, NULL);
```

jest równoznaczne poniższemu wywołaniu funkcji `waitpid()`:

```
pid = waitpid (-1, status, options);
```

Natomiast wywołanie funkcji `wait4()`:

```
pid = wait4 (pid, status, options, NULL);
```

jest równoznaczne wywołaniu funkcji `waitpid()`:

```
pid = waitpid (pid, status, options);
```

Oznacza to, że funkcja `wait3()` oczekuje na zmianę stanu dla dowolnego procesu potomnego, natomiast `wait4()` oczekuje na zmianę stanu dla określonego potomka, którego identyfikator procesu równy jest parametrowi `pid`. Parametr `options` działa tak samo jak w przypadku funkcji `waitpid()`.

Jak wcześniej wspomniano, duża różnica między tymi funkcjami a funkcją `waitpid()` spowodowana jest zastosowaniem parametru `rusage`. Jeśli nie jest on równy `NULL`, wówczas funkcja odpowiednio uaktualnia strukturę wskazywaną przez wskaźnik `rusage`, wpisując w niej informacje dotyczące procesu potomnego. Struktura `rusage` dostarcza informacji dotyczących użycia zasobów przez proces potomny:

```
#include <sys/resource.h>
struct rusage
{
   struct timeval ru_utime; /* zużyty czas użytkownika */
   struct timeval ru_stime; /* zużyty czas systemowy */
   long ru_maxrss; /* maksymalny rozmiar grupy rezydenckiej */
   long ru_ixrss; /* wielkość pamięci udostępnianej */
   long ru_idrss; /* wielkość danych nieudostępnianych */
   long ru_isrss; /* rozmiar stosu nieudostępnianego */
   long ru_minflt; /* liczba odzyskań stron */
   long ru_majflt; /* liczba błędów stron */
   long ru_nswap; /* liczba operacji przerzucania pamięci na dysk */
   long ru_inblock; /* liczba operacji blokowania wejścia */
   long ru_oublock; /* liczba operacji blokowania wyjścia */
   long ru_msgsnd; /* liczba wiadomości wysłanych */
   long ru_msgrcv; /* liczba wiadomości odebranych */
   long ru_nsignals; /* liczba odebranych sygnałów */
   long ru_nvcsw; /* liczba dobrowolnych przełączeń kontekstu */
   long ru_nivcsw; /* liczba wymuszonych przełączeń kontekstu */
};
```

W następnym rozdziale będzie kontynuowany temat użycia zasobów.

W przypadku sukcesu funkcje te zwracają PID dla procesu, którego status uległ zmianie. W przypadku błędu zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na te same wartości błędu jak w przypadku funkcji `waitpid()`.

Ponieważ funkcje `wait3()` oraz `wait4()` nie są zdefiniowane w standardzie POSIX[6], zaleca się, aby używać ich wyłącznie wtedy, gdy najważniejszy staje się dostęp do informacji związanych z użyciem zasobów. Pomimo tego że POSIX nie wspiera tych funkcji, są one jednak zaimplementowane w prawie każdej wersji systemu Unix.

Uruchamianie i oczekiwanie na nowy proces

Zarówno standard ANSI C, jak i POSIX definiują interfejs, który łączy operacje uruchamiania nowego procesu oraz oczekiwania na jego zakończenie — można powiedzieć, że służy on do synchronicznego utworzenia procesu. Jeśli dany proces tworzy swojego potomka i natychmiast chce oczekiwać na jego zakończenie, wówczas sensowne staje się użycie następującego interfejsu:

```
#define _XOPEN_SOURCE /* w przypadku użycia makra WEXITSTATUS itd. */
#include <stdlib.h>

int system (const char *command);
```

Funkcja `system()` posiada taką właśnie nazwę, ponieważ synchroniczne wywołanie procesu zwane jest *wejściem do powłoki systemowej*. Powszechnie używa się tej funkcji, aby uruchamiać proste programy użytkowe lub skrypty powłoki, oczekując często na kod powrotu z takich aplikacji.

Wywołanie funkcji `system()` powoduje wykonanie polecenia przekazanego w parametrze `command`, włącznie z użyciem jego dodatkowych argumentów. Łańcuch znaków, zawierający parametr `command`, zostaje dołączony do polecenia */bin/sh -c*. Parametr ten zostaje całkowicie przekazany do powłoki systemowej.

W przypadku sukcesu wartością powrotną funkcji jest kod powrotu danego polecenia, podobnie jak w przypadku funkcji `wait()`. Zgodnie z tym kod powrotu wykonanego polecenia można otrzymać poprzez użycie makra `WEXITSTATUS`. Jeśli wywołanie samej powłoki */bin/sh* nie powiedzie się, makro `WEXITSTATUS` zwróci taką samą wartość jak w przypadku wykonania funkcji `exit(127)`. Ponieważ jest możliwe, aby wykonywane polecenie również zwracało wartość `127`, dlatego też nie istnieje niezawodna metoda na sprawdzenie, czy to rzeczywiście powłoka zwróciła taki błąd. W przypadku błędu funkcja zwraca `-1`.

Jeśli parametr `command` równy jest `NULL`, funkcja `system()` zwraca wartość niezerową, gdy istnieje powłoka */bin/sh*; w przeciwnym razie zwraca zero.

Podczas wykonywania polecenia sygnał `SIGCHLD` zostaje zablokowany, natomiast sygnały `SIGINT` oraz `SIGQUIT` są ignorowane. Ignorowanie sygnałów `SIGINT` i `SIGQUIT` łączy się z pewnymi konsekwencjami, szczególnie, jeśli funkcja `system()` jest wywoływana w pętli. Jeśli jest wywoływana z wnętrza pętli, wówczas należy się upewnić, że program poprawnie sprawdza kod powrotu procesu potomnego. Na przykład:

```
do
{
   int ret;

   ret = system ("pidof rudderd");
   if (WIFSIGNALED (ret) && (WTERMSIG (ret) == SIGINT || WTERMSIG (ret) == SIGQUIT))
      break; /* lub inny rodzaj obsługi */
} while (1);
```

[6] Funkcja `wait3()` została dołączona do pierwotnej specyfikacji *Single UNIX Specification*, lecz następnie została z niej usunięta.

Użytecznym ćwiczeniem jest zaimplementowanie funkcji `system()` przy użyciu `fork()`, funkcji z grupy *exec* oraz funkcji `waitpid()`. Programista systemowy powinien spróbować zrealizować to zadanie samodzielnie —łączy ono w sobie wiele tematów omówionych w tym rozdziale. Aby zapewnić kompletność informacji, poniżej przedstawiono prostą implementację:

```
/*
* my_system - synchronicznie uruchamia i czeka na wykonanie polecenia
* "/bin/sh -c <cmd>".
*
* W przypadku dowolnego błędu zwraca -1 lub kod powrotu
* dla uruchomionego procesu. Nie blokuje, ani nie ignoruje żadnych sygnałów.
*/
int my_system (const char *cmd)
{
   int status;
   pid_t pid;

   pid = fork ( );
   if (pid == -1)
      return -1;
   else if (pid == 0)
   {
      const char *argv[4];

      argv[0] = "sh";
      argv[1] = "-c";
      argv[2] = cmd;
      argv[3] = NULL;
      execv ("/bin/sh", argv);

      exit (-1);
   }
   if (waitpid (pid, &status, 0) == -1)
      return -1;
   else if (WIFEXITED (status))
      return WEXITSTATUS (status);

   return -1;
}
```

Należy zwrócić uwagę, że w przeciwieństwie do oryginalnej funkcji `system()` ten przykład nie blokuje, ani nie wyłącza żadnych sygnałów. To zachowanie może być lepsze lub gorsze, w zależności od sytuacji w programie, lecz mimo to często dobrą decyzją jest brak blokady przynajmniej dla sygnału `SIGINT`, ponieważ przerywa wykonywane polecenia w taki sposób, jak tego zwykle oczekują użytkownicy. Lepsza implementacja mogłaby dodać kolejne parametry wskaźnikowe, które (gdyby nie były równe `NULL`) oznaczałyby pojawianie się różnych błędów. Na przykład, można by dodać kody błędów `fork_failed` i `shell_failed`.

Procesy zombie

Jak wcześniej napisano, proces, który został zakończony, ale jeszcze nieobsłużony przez swojego rodzica, nazywany jest procesem zombie. Procesy zombie wciąż zużywają zasoby systemowe, choć w niewielkiej części — wystarczającej jednak, by obsłużyć minimalny szkielet pozostały po tym, czym kiedyś były. Te pozostawione zasoby są potrzebne, aby procesy rodzicielskie, które chcą sprawdzić status swoich procesów potomnych, mogły otrzymać odpowiednie informacje dotyczące ich działania i zakończenia. Gdy tylko potomek zostanie w ten sposób obsłużony przez rodzica, jądro usuwa go zupełnie i proces zombie przestaje istnieć.

Jednak każdy, kto kiedykolwiek używał systemu Unix, jest pewien, że widział bezczynne procesy zombie. Procesy te, zwane często *duchami*, posiadają „nieodpowiedzialnych" rodziców. Jeśli aplikacja tworzy nowy proces, jest odpowiedzialna (chyba że jej czas istnienia jest krótki) za obsłużenie procesu potomnego, nawet jeśli pozbędzie się otrzymanych od niego informacji. W przeciwnym razie wszystkie procesy potomne staną się duchami i będą w dalszym ciągu istnieć, wydłużając listę procesów w systemie i sprawiając wrażenie, że aplikacja została napisana niestarannie.

Co jednak stanie się, jeśli proces rodzicielski zostanie zakończony przed procesem potomnym lub jeśli zakończy swoje istnienie, zanim będzie miał szansę, aby obsłużyć swoje procesy zombie? Gdy następuje zakończenie dowolnego procesu, jądro Linuksa przetwarza całą listę jego procesów potomnych, zmieniając im rodzica na proces inicjalizujący (proces, który posiada PID równy 1). W ten sposób zapewnia, że w systemie nie będzie w ogóle procesów, które nie posiadają bezpośrednich rodziców. Z kolei proces inicjalizujący regularnie obsługuje wszystkie swoje procesy potomne, zapewniając, że żaden z nich nie pozostanie w stanie zombie zbyt długo, co oznacza, że w systemie nie ma duchów! Dlatego też, jeśli proces rodzicielski zostanie zakończony przed swoimi procesami potomnymi lub jeśli nie obsłuży swoich potomków przed zakończeniem swojego istnienia jako rodzic, procesom potomnym zostanie w końcu przypisany proces inicjalizujący, który będzie je obsługiwać, pozwalając im poprawnie się zakończyć. To zabezpieczenie oznacza, że w przypadku procesów o krótkim czasie istnienia nie należy zbytnio przejmować się obsługiwaniem wszystkich ich potomków — taki sposób postępowania jest wciąż uważany za poprawny.

Użytkownicy i grupy

Jak wspomniano wcześniej w tym rozdziale, a także omówiono dokładniej w rozdziale 1., procesy związane są z użytkownikami i grupami. Identyfikatory użytkownika i grupy są wartościami liczbowymi reprezentowanymi przez odpowiednie typy języka C: `uid_t` oraz `gid_t`. Odwzorowanie pomiędzy wartościami liczbowymi a nazwami, przyjaznymi dla ludzi (jak na przykład UID równy zeru, przypisany użytkownikowi *root*), wykonywane jest w przestrzeni użytkownika przy użyciu plików */etc/passwd* oraz */etc/group*. Jądro obsługuje wyłącznie wartości liczbowe.

Właściciel oraz grupa właścicielska procesu definiują zakres działań, które ten proces może wykonywać w systemie Linux. Procesy muszą być uruchamiane „w imieniu" odpowiednich użytkowników lub grup. Wiele procesów przypisanych jest użytkownikowi administracyjnemu (*root*). Jednakże podczas tworzenia oprogramowania najbardziej zalecanym sposobem postępowania jest stosowanie zasady przydzielania *najmniej uprzywilejowanych* uprawnień, oznaczającej, że proces powinien być uruchamiany z możliwie najmniejszym poziomem uprawnień. Wymaganie to jest elastyczne: jeśli proces wymaga uprawnień administratora, aby na początku swojego istnienia przeprowadzić pewną operację, lecz później już ich nie potrzebuje, wówczas jak najszybciej powinien się ich pozbyć. Aby to uzyskać, wiele procesów często manipuluje swoimi identyfikatorami użytkownika i grupy.

Zanim zostanie pokazane, w jaki sposób ta manipulacja może być zrealizowana, należy najpierw omówić zawiłości definicji identyfikatorów użytkownika i grupy.

Rzeczywiste, efektywne oraz zapisane identyfikatory użytkownika i grupy

Aktualna analiza dotyczy identyfikatorów użytkownika, lecz w przypadku identyfikatorów grupy sytuacja jest taka sama.

W rzeczywistości istnieje nie jeden, lecz cztery rodzaje identyfikatorów użytkownika związanych z procesem: rzeczywisty UID, efektywny UID, zapisany UID oraz UID dla systemu plików. *Rzeczywisty identyfikator użytkownika* jest identyfikatorem użytkownika, który pierwotnie uruchomił proces. Zostaje on ustawiony na wartość faktycznego identyfikatora użytkownika dla rodzica tego procesu i nie zmienia się podczas wykonywania funkcji *exec*. Zwykle proces logowania przypisuje rzeczywisty identyfikator użytkownika dla powłoki temu użytkownikowi, który przy pomocy niego loguje się do systemu. Wszystkie procesy, uruchomione później przez proces logowania, będą posiadać UID zalogowanego użytkownika. Użytkownik administracyjny (*root*) może zmienić rzeczywisty identyfikator użytkownika na dowolną wartość; nie może tego wykonać nikt inny.

Efektywny identyfikator użytkownika jest identyfikatorem użytkownika, którego aktualnie używa dany proces. Ta wartość jest zwykle wykorzystywana w procedurach kontroli uprawnień. Na początku identyfikator ten równy jest rzeczywistemu identyfikatorowi użytkownika, ponieważ w momencie rozwidlenia procesu odziedziczony zostaje efektywny identyfikator użytkownika dla rodzica. Później, gdy proces wywołuje funkcję *exec*, efektywny identyfikator użytkownika pozostaje bez zmian. Jednak podczas wykonywania tej funkcji pojawia się zasadnicza różnica między rzeczywistym, a efektywnym identyfikatorem użytkownika: poprzez uruchomienie obrazu binarnego, posiadającego ustawiony bit uprawnień *setuid* (*suid*), proces może zmienić swój efektywny UID. Efektywny identyfikator użytkownika zostaje ustawiony na identyfikator użytkownika będącego właścicielem pliku programu. Na przykład, ponieważ plik */usr/bin/passwd* jest plikiem z uprawnieniem *setuid*, a jego właścicielem jest administrator, dlatego też, gdy powłoka zwykłego użytkownika tworzy proces uruchamiający ten plik, następuje ustawienie efektywnego identyfikatora użytkownika na wartość równą identyfikatorowi administratora, a nie użytkownika, który go uruchamia.

Użytkownicy zwykli mogą ustawiać efektywny UID na wartość rzeczywistego lub zapisanego identyfikatora użytkownika. Administrator może ustawiać efektywny identyfikator użytkownika na dowolną wartość.

Zapisany identyfikator użytkownika jest pierwotnym efektywnym identyfikatorem użytkownika dla danego procesu. Gdy następuje rozwidlenie procesu, potomek dziedziczy od swojego rodzica zapisany identyfikator użytkownika. Jednak po uruchomieniu funkcji *exec* jądro ustawia wartość zapisanego identyfikatora użytkownika na efektywny identyfikator użytkownika, dzięki czemu zapamiętuje go na czas trwania wywołania funkcji *exec*. Użytkownicy zwykli nie mogą modyfikować zapisanego identyfikatora użytkownika; administrator może zmieniać go na taką samą wartość jak w przypadku rzeczywistego identyfikatora użytkownika.

Jakie jest zastosowanie tych wszystkich wartości? Efektywny identyfikator użytkownika jest wartością, która ma duże znaczenie. Jest to identyfikator użytkownika, brany pod uwagę podczas kontroli poprawności dla uwierzytelnień procesu. Rzeczywisty identyfikator użytkownika oraz

zapisany identyfikator użytkownika spełniają rolę zastępczych, ewentualnych identyfikatorów użytkownika, które mogą być wykorzystywane przez zwykłe procesy bez uprawnień administratora. Rzeczywisty identyfikator użytkownika jest efektywnym UID należącym do użytkownika, który faktycznie uruchamia program. Natomiast zapisany identyfikator użytkownika jest posiadanym przez proces efektywnym identyfikatorem użytkownika do momentu, gdy plik binarny z uprawnieniami *suid* spowodował jego zmianę podczas wykonywania funkcji *exec*.

Zmiana rzeczywistego lub zapisanego identyfikatora dla użytkownika lub grupy

Identyfikatory użytkownika i grupy ustawiane są za pomocą dwóch funkcji systemowych:

```
#include <sys/types.h>
#include <unistd.h>

int setuid (uid_t uid);
int setgid (gid_t gid);
```

Wywołanie funkcji `setuid()` ustawia efektywny identyfikator użytkownika dla danego procesu. Jeśli aktualny efektywny identyfikator użytkownika dla procesu równy jest zeru (*root*), wówczas zostają również ustawione rzeczywisty i zapisany identyfikator użytkownika. Użytkownik administracyjny może przekazywać dowolną wartość w parametrze `uid`, dzięki czemu możliwe jest zainicjalizowanie nią wszystkich trzech rodzajów identyfikatorów użytkownika. Użytkownik zwykły ma jedynie możliwość przekazania rzeczywistego lub zapisanego identyfikatora użytkownika w parametrze `uid`. Inaczej mówiąc, użytkownik zwykły może ustawić wyłącznie efektywny identyfikator użytkownika na jedną z tych wartości.

W przypadku sukcesu, funkcja `setuid()` zwraca wartość zero. W przypadku błędu, zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EAGAIN`
: Wartość przekazana w parametrze `uid` różni się od rzeczywistego identyfikatora użytkownika. Podanie rzeczywistego identyfikatora użytkownika w parametrze `uid` powoduje przekroczenie limitu `NPROC` dla danego procesu (limit ten określa liczbę procesorów, do których ma dostęp użytkownik).

`EPERM`
: Użytkownik nie ma uprawnień administracyjnych, a parametr `uid` nie jest ani efektywnym, ani zapisanym identyfikatorem użytkownika.

Powyższe informacje dotyczą również grup — należy po prostu zamienić `setuid()` z `setgid()`, a `uid` z `gid`.

Zmiana efektywnego identyfikatora dla użytkownika lub grupy

Linux dostarcza dwóch funkcji, zdefiniowanych przez POSIX, które pozwalają na ustawianie efektywnego identyfikatora użytkownika lub grupy dla aktualnie działającego procesu:

```
#include <sys/types.h>
#include <unistd.h>

int seteuid (uid_t euid);
int setegid (gid_t egid);
```

Wywołanie funkcji `seteuid()` ustawia efektywny identyfikator użytkownika na wartość przekazaną w parametrze `euid`. Użytkownik administracyjny może przekazywać dowolną wartość w tym parametrze. W przypadku użytkowników zwykłych efektywny identyfikator użytkownika może być równy jedynie wartości rzeczywistego lub zapisanego UID. W przypadku sukcesu funkcja `seteuid()` zwraca `0`. W przypadku błędu, zwraca `-1` oraz ustawia zmienną `errno` na wartość `EPERM`, która oznacza, że właścicielem aktualnego procesu nie jest administrator, a parametr `euid` nie jest równy ani rzeczywistemu, ani zapisanemu identyfikatorowi użytkownika.

Należy zwrócić uwagę na to, że w przypadku użytkowników zwykłych funkcje `seteuid()` oraz `setuid()` zachowują się tak samo. Jest to dobre rozwiązanie i powinno być standardową procedurą, aby zawsze używać funkcji `seteuid()`, chyba że proces wymaga praw administratora — w tym przypadku większy sens ma użycie funkcji `setuid()`.

Powyższe informacje dotyczą również grup — należy po prostu zamienić `seteuid()` z `setegid()`, a `euid` z `egid`.

Zmiana identyfikatora dla użytkownika lub grupy w wersji BSD

W systemie operacyjnym BSD stworzono własne interfejsy pozwalające na ustawianie identyfikatorów użytkownika i grupy. Linux udostępnia te interfejsy ze względów na kompatybilność:

```
#include <sys/types.h>
#include <unistd.h>

int setreuid (uid_t ruid, uid_t euid);
int setregid (gid_t rgid, gid_t egid);
```

Wywołanie funkcji `setreuid()` ustawia rzeczywisty i efektywny identyfikator użytkownika dla procesu na wartości odpowiednio równe parametrom `ruid` i `euid`. Przekazanie `-1` w którymś z tych parametrów pozostawia niezmienioną wartość odpowiedniego identyfikatora użytkownika. Użytkownicy zwykli mogą jedynie ustawiać efektywny identyfikator użytkownika na wartość rzeczywistego lub zapisanego identyfikatora użytkownika, a rzeczywisty identyfikator użytkownika na wartość efektywnego identyfikatora użytkownika. Jeśli zostanie zmieniony rzeczywisty identyfikator użytkownika lub jeśli efektywny identyfikator użytkownika zostanie zmieniony na wartość różną od poprzedniego rzeczywistego identyfikatora użytkownika, wówczas zapisany identyfikator użytkownika będzie zmieniony na wartość nowego efektywnego identyfikatora użytkownika. Przynajmniej w taki sposób reaguje na te zmiany Linux i większość innych systemów uniksowych; zachowanie to nie jest zdefiniowane w standardzie POSIX.

W przypadku sukcesu, funkcja `setreuid()` zwraca `0`. W przypadku błędu, zwraca `-1` oraz ustawia zmienną `errno` na wartość `EPERM`, która oznacza, że właścicielem aktualnego procesu nie jest administrator, a parametr `euid` nie jest równy ani rzeczywistemu, ani zapisanemu identyfikatorowi użytkownika lub parametr `ruid` nie jest równy efektywnemu identyfikatorowi użytkownika.

Powyższe informacje dotyczą również grup — należy po prostu zamienić `setreuid()` z `setre`
`↪gid()`, `ruid` z `rgid`, a `euid` z `egid`.

Zmiana identyfikatora dla użytkownika lub grupy w wersji HP-UX

Wygląda na to, że sytuacja zaczyna wymykać się spod kontroli, gdyż w systemie HP-UX (systemie Unix firmy Hewlett-Packard) również zaimplementowano swoje własne mechanizmy, pozwalające na ustawianie identyfikatorów użytkownika i grupy dla procesu. Linux nadąża jednak za tymi zmianami i udostępnia następujące interfejsy:

```
#define _GNU_SOURCE
#include <unistd.h>

int setresuid (uid_t ruid, uid_t euid, uid_t suid);
int setresgid (gid_t rgid, gid_t egid, gid_t sgid);
```

Wywołanie funkcji `setresuid()` ustawia rzeczywisty, efektywny i zapisany identyfikator użytkownika na wartości odpowiednio równe parametrom `ruid`, `euid` oraz `suid`. Przekazanie `-1` w którymś z tych parametrów pozostawia niezmienioną wartość odpowiedniego identyfikatora użytkownika.

Użytkownicy z prawami administratora mogą ustawiać każdy identyfikator użytkownika na dowolną wartość. Użytkownicy zwykli mogą ustawiać każdy identyfikator użytkownika na wartość rzeczywistego, efektywnego lub zapisanego identyfikatora użytkownika W przypadku sukcesu, funkcja `setresuid()` zwraca wartość zero. W przypadku błędu, zwraca `-1` oraz ustawia zmienną `errno` na jedną z poniższych wartości:

`EAGAIN`
: Wartość przekazana w parametrze `uid` różni się od rzeczywistego identyfikatora użytkownika. Podanie rzeczywistego identyfikatora użytkownika w parametrze `uid` powoduje przekroczenie limitu `NPROC` dla danego procesu (limit ten określa liczbę procesorów, do których ma dostęp użytkownik).

`EPERM`
: Użytkownik nie ma uprawnień administracyjnych i nastąpiła próba ustawienia nowych wartości dla rzeczywistego, efektywnego lub zapisanego identyfikatora użytkownika, które nie pasują do aktualnych wartości tych identyfikatorów.

Powyższe informacje dotyczą również grup — należy po prostu zamienić `setresuid()` z `setres`↪`gid()`, `ruid` z `rgid`, `euid` z `egid`, a `suid` z `sgid`.

Zalecane modyfikacje identyfikatorów użytkownika i grupy

Procesy zwykłe powinny użyć funkcji `seteuid()`, aby zmienić swój efektywny identyfikator użytkownika. Procesy administratora powinny używać funkcji `setuid()`, jeśli muszą zmienić wszystkie trzy identyfikatory użytkownika, natomiast funkcji `seteuid()`, jeśli potrzebują tymczasowo zmienić tylko efektywny UID. Funkcje te są proste i działają zgodnie ze standardem POSIX, biorąc pod uwagę również zapisane identyfikatory użytkownika.

Pomimo dodatkowej funkcjonalności, wersje tych funkcji, pochodzące z systemów BSD oraz HP-UX, nie wprowadzają żadnych nowych i użytecznych opcji, których nie posiadałyby już funkcje `setuid()` i `seteuid()`.

Wsparcie dla zapisanych identyfikatorów użytkownika

Istnienie zapisanych identyfikatorów użytkownika i grupy zdefiniowane zostało w standardzie IEEE Std 1003.1-2001 (POSIX 2001), a Linux wspierał te identyfikatory od dawna, poczynając od wersji jądra 1.1.38. W przypadku programów napisanych wyłącznie dla Linuksa, można być pewnym, że zapisane identyfikatory użytkownika istnieją w systemie. W programach napisanych dla starszych wersji Uniksa, zanim przystąpi się do wykonywania modyfikacji związanych z zapisanymi identyfikatorami użytkownika lub grupy, należy sprawdzić, czy istnieje makro o nazwie `_POSIX_SAVED_IDS`.

W przypadku braku wsparcia dla zapisanych identyfikatorów użytkownika i grupy, poprzednie rozważania są w dalszym ciągu poprawne; wystarczy po prostu pominąć wszystkie fragmenty tekstu, które dotyczą takich identyfikatorów.

Otrzymywanie identyfikatorów użytkownika i grupy

Istnieją dwie funkcje systemowe, które zwracają odpowiednio rzeczywiste identyfikatory użytkownika oraz grupy:

```
#include <unistd.h>
#include <sys/types.h>

uid_t getuid (void);
gid_t getgid (void);
```

Funkcje te nie mogą wykonać się niepoprawnie. Poniższe dwie funkcje systemowe zwracają odpowiednio efektywne identyfikatory użytkownika oraz grupy:

```
#include <unistd.h>
#include <sys/types.h>

uid_t geteuid (void);
gid_t getegid (void);
```

Te funkcje również nie mogą wykonać się niepoprawnie.

Grupy sesji i procesów

Każdy proces należy do *grupy procesów*, która jest zbiorem jednego lub więcej procesów, zazwyczaj powiązanych ze sobą, aby umożliwić *zarządzanie zadaniami*. Główną cechą tej grupy jest to, że do wszystkich procesów, które do niej należą, mogą być wysyłane sygnały: pojedynczy sygnał może zakończyć, zatrzymać lub ponownie uruchomić wszystkie procesy z tej samej grupy procesów.

Każda grupa procesów rozróżniana jest poprzez unikalny *identyfikator grupy procesów* (PGID) oraz posiada *lidera grupy procesów*. Identyfikator grupy procesów równy jest identyfikatorowi użytkownika, który posiada lider grupy procesów. Grupa procesów istnieje tak długo, dopóki składa się przynajmniej z jednego elementu. Nawet jeśli proces lidera grupy procesów zostaje zakończony, grupa procesów w dalszym ciągu istnieje.

Gdy do maszyny po raz pierwszy loguje się nowy użytkownik, proces logowania tworzy nową *sesję* składającą się z jednego procesu, którym jest *domyślna powłoka* użytkownika. Domyślna powłoka funkcjonuje jako *lider sesji*. Identyfikator użytkownika dla lidera sesji nazywany jest

identyfikatorem sesji. Sesja jest zbiorem jednej lub więcej grup procesów. Sesje udostępniają użytkownikowi obsługę i możliwość logowania się do systemu, a także wiążą go z *terminalem sterującym*. Terminal sterujący jest określonym urządzeniem tty, obsługującym terminalowe operacje wejścia i wyjścia dla użytkownika. Zgodnie z tym sesje są w dużym stopniu związane z powłoką. Właściwie żaden inny element systemu nie ma powiązania z sesjami.

Grupa procesów dostarcza mechanizmu pozwalającego przesyłać sygnały do wszystkich jej elementów i dzięki temu usprawniać zarządzanie zadaniami oraz innymi funkcjami powłoki, natomiast sesje istnieją, aby łączyć procesy logowania się użytkowników z terminalami sterującymi. Grupy procesów w sesji składają się z jednej *pierwszoplanowej grupy procesów* oraz zera lub więcej *drugoplanowych grup procesów*. Gdy użytkownik odłącza się od terminala, następuje wysłanie sygnału `SIGQUIT` do wszystkich procesów należących do drugoplanowych grup procesów. Gdy terminal wykrywa zerwanie połączenia sieciowego, następuje wysłanie sygnału `SIGHUP` również do tych samych grup procesów. Gdy użytkownik naciska sekwencję klawiszy przerwania (zwykle *Ctrl*+*C*), sygnał `SIGINT` zostaje wysłany do wszystkich procesów, należących także do drugoplanowych grup procesów. Dlatego też sesje pozwalają na łatwiejsze zarządzanie terminalami i procesami logowania przez powłoki systemowe.

Na przykład, załóżmy, że użytkownik loguje się do systemu, a jego domyślna powłoka, którą jest program *bash*, posiada identyfikator użytkownika o numerze 1700. Egzemplarz powłoki *bash* dla użytkownika jest obecnie jedynym elementem oraz liderem w nowej grupie procesów, która posiada identyfikator 1700. Ta grupa procesów należy do sesji o identyfikatorze 1700, natomiast powłoka *bash* jest jedynym elementem i liderem w tej sesji. Nowe polecenia, które są uruchamiane przez użytkownika w powłoce systemowej, powodują powstanie nowych grup procesów dla sesji 1700. Jedna z tych grup procesów — ta, która jest bezpośrednio połączona z użytkownikiem i steruje terminalem — jest pierwszoplanową grupą procesów. Wszystkie inne grupy procesów są drugoplanowe.

W danym systemie istnieje wiele sesji: sesje logowania (po jednej dla każdego użytkownika) oraz inne, dla procesów niezwiązanych z logowaniem, takich jak demony. Demony często posiadają swoje własne sesje, aby chronić się przed problemami powstającymi podczas wiązania się z sesjami, które mogą ulec przedwczesnemu zakończeniu.

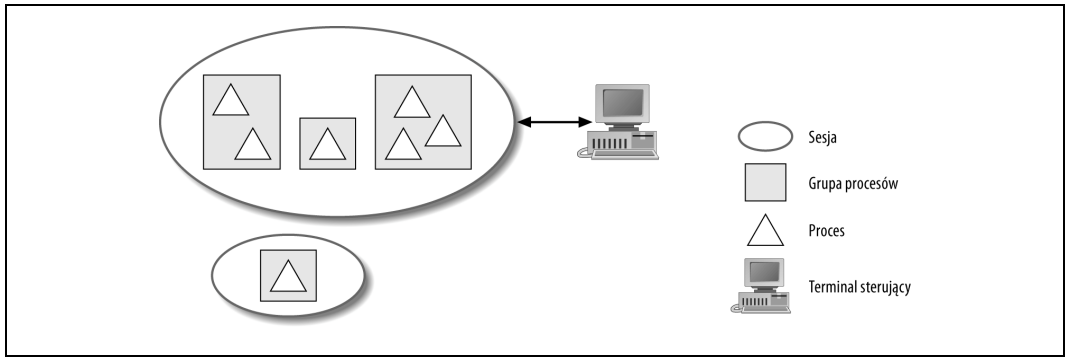
Każda z sesji zawiera jedną lub więcej grup procesów, a każda grupa procesów zawiera przynajmniej jeden proces. Grupy procesów, które zawierają więcej niż jeden proces, zazwyczaj implementują mechanizm zarządzania zadaniami.

Oto przykładowe polecenie powłoki:

```
$ cat ship-inventory.txt | grep booty | sort
```

Polecenie to spowoduje powstanie jednej grupy procesów, zawierającej trzy procesy. Dzięki temu powłoka może przesyłać sygnały do wszystkich trzech procesów jednocześnie. Ponieważ użytkownik wpisał w konsoli powyższy tekst bez zakończenia go znakiem `&`, można również powiedzieć, że ta grupa sesji jest pierwszoplanowa. Rysunek 5.1. przedstawia powiązania między sesjami, grupami procesów, procesami i terminalami sterującymi.

Linux dostarcza kilku interfejsów w celu ustawiania i odzyskiwania identyfikatorów sesji oraz grupy procesów związanej z danym procesem. Są one użyteczne przede wszystkim dla powłoki systemowej, lecz mogą również przydać się w przypadku takich procesów jak demony, dla których należy tworzyć własną obsługę sesji i grup procesów.



Rysunek 5.1. Powiązania między sesjami, grupami procesów, procesami i terminalami sterującymi

Funkcje systemowe do obsługi sesji

Powłoka tworzy nowe sesje podczas logowania. Wykonuje to za pomocą specjalnej funkcji systemowej, która ułatwia tworzenie nowej sesji:

```
#include <unistd.h>

pid_t setsid (void);
```

Wywołanie funkcji `setsid()` powoduje utworzenie nowej sesji przy założeniu, że proces nie jest jeszcze liderem grupy procesów. Proces wywołujący staje się liderem sesji i jej jedynym elementem. Sesja ta nie posiada terminala sterującego. Funkcja `setsid()` tworzy także dla tej sesji nową grupę procesów oraz powoduje, że proces wywołujący staje się również jej liderem. Jest to użyteczne zachowanie dla demonów, które nie powinny należeć do żadnej istniejącej sesji lub posiadać terminali sterujących, jak również dla powłok, dla których wymagane jest tworzenie nowej sesji dla każdego użytkownika po jego zalogowaniu się w systemie.

W przypadku sukcesu funkcja `setsid()` zwraca identyfikator sesji, która właśnie została utworzona. W przypadku błędu zwraca `−1`. Jedynym możliwym kodem błędu jest `EPERM`. Oznacza on, że proces jest obecnie liderem grupy procesów. Najprostszym sposobem na upewnienie się, że dany proces nie jest liderem grupy procesów, jest wykonanie rozwidlenia, zakończenie procesu rodzicielskiego oraz uruchomienie funkcji `setsid()` przez proces potomny. Na przykład:

```
pid_t pid;

pid = fork ( );
if (pid == -1)
{
   perror ("fork");
   return -1;
}
else if (pid != 0)
   exit (EXIT_SUCCESS);

if (setsid ( ) == -1)
{
   perror ("setsid");
   return -1;
}
```

Możliwe jest również odczytywanie aktualnego identyfikatora sesji, który jest jednak mniej przydatny:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getsid (pid_t pid);
```

Wywołanie funkcji `getsid()` zwraca identyfikator sesji dla procesu określonego w parametrze `pid`. Jeśli parametr `pid` równy jest zeru, funkcja zwraca identyfikator sesji dla procesu wywołującego. W przypadku błędu, zwraca `-1`. Jedyną możliwą wartością zmiennej `errno` jest `ESRCH`, która oznacza, że parametr `pid` nie odpowiada identyfikatorowi żadnego istniejącego procesu. Należy zauważyć, że inne systemy uniksowe mogą ustawiać zmienną `errno` na wartość `EPERM`, oznaczającą, że parametr `pid` oraz proces wywołujący nie należą do tej samej sesji; Linux nie zwraca tego błędu i szczęśliwie udostępnia identyfikatory sesji dla każdego procesu.

Funkcja używana jest rzadko i raczej dla celów diagnostycznych:

```
pid_t sid;

sid = getsid (0);
if (sid == -1)
   perror ("getsid"); /* ten błąd nie powinien wystąpić */
else
   printf ("Identyfikator mojej sesji = %d\n", sid);
```

Funkcje systemowe do obsługi grup procesów

Wywołanie funkcji `setpgid()` powoduje ustawienie identyfikatora grupy procesów dla procesu, określonego w parametrze `pid`, na wartość `pgid`:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);
```

Jeśli w parametrze `pid` przekazane zostanie zero, użyty zostanie aktualny proces. Jeśli `pgid` wynosi zero, jako identyfikator grupy procesu zostanie użyty identyfikator procesu przekazany w parametrze `pid`.

W przypadku sukcesu, funkcja `setpgid()` zwraca `0`. Sukces jest zależny od paru warunków:

- Proces, przekazany w parametrze `pid`, musi być procesem wywołującym lub jego potomkiem, który nie wywołał funkcji *exec* i należy do tej samej sesji co proces wywołujący.
- Proces, przekazany w parametrze `pid`, nie może być liderem sesji.
- Jeśli istnieje już proces, określony w parametrze `pgid`, wówczas musi należeć do tej samej sesji co proces wywołujący.
- Parametr `pgid` musi posiadać wartości nieujemne.

W przypadku błędu funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces o identyfikatorze, przekazanym w parametrze `pid`, jest potomkiem procesu wywołującego, który wykonał już funkcję *exec*.

EINVAL
: Parametr pgid jest mniejszy od zera.

EPERM
: Proces o identyfikatorze, przekazanym w parametrze pid, jest liderem sesji lub należy do innej sesji niż proces wywołujący. Alternatywnym powodem tego błędu może być próba przeniesienia procesu do grupy procesów, należącej do innej sesji.

ESRCH
: Parametr pid nie jest aktualnym procesem, potomkiem aktualnego procesu lub nie jest równy zeru.

Podobnie jak w przypadku sesji, uzyskiwanie identyfikatora grupy procesów dla procesu jest możliwe, lecz mniej przydatne:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getpgid (pid_t pid);
```

Wywołanie funkcji getpgid() zwraca identyfikator grupy procesów dla procesu wskazywanego w parametrze pid. Jeśli pid równy jest zeru, zwrócony zostaje identyfikator grupy procesów dla aktualnego procesu. W przypadku błędu, funkcja zwraca −1 oraz ustawia zmienną errno na jedyną możliwą wartość ESRCH, która oznacza, że pid jest niepoprawnym identyfikatorem procesu.

Podobnie jak w przypadku funkcji getsid(), ta funkcja używana jest raczej dla celów diagnostycznych:

```
pid_t pgid;

pgid = getpgid (0);
if (pgid == -1)
   perror ("getpgid"); /* ten błąd nie powinien wystąpić */
else
   printf ("Identyfikator mojej grupy procesów = %d\n", pgid);
```

Przestarzałe funkcje do obsługi grupy procesów

Linux dostarcza dwóch starszych interfejsów, pochodzących z systemu BSD, które służą do modyfikacji oraz uzyskiwania identyfikatora grupy procesów. Ponieważ są one mniej przydatne niż poprzednio omówione funkcje systemowe, dlatego też w przypadku nowych programów powinno używać się ich jedynie wtedy, gdy wymagane jest rygorystyczne spełnienie warunków przenośności. Funkcja setpgrp() może zostać użyta w celu ustawienia identyfikatora grupy procesów:

```
#include <unistd.h>

int setpgrp (void);
```

Rozważmy poniższe wywołanie:

```
if (setpgrp ( ) == -1)
   perror ("setpgrp");
```

Jest ono identyczne z następującym wywołaniem:

```
if (setpgid (0,0) == -1)
   perror ("setpgid");
```

Obie funkcje próbują przypisać aktualny proces do grupy procesów o tym samym numerze jak jego identyfikator. W przypadku sukcesu, funkcje zwracają `0`, natomiast w przypadku błędu zwracają `−1`. Prawie wszystkie wartości zmiennej `errno`, należące do funkcji `setpgid()`, mogą zostać zastosowane w przypadku `setpgrp()`. Wyjątkiem jest wartość `ESRCH`.

Analogicznie, wywołanie funkcji `getpgrp()` może zostać użyte w celu uzyskania identyfikatora grupy procesów:

```
#include <unistd.h>

pid_t getpgrp (void);
```

Weźmy pod uwagę poniższe wywołanie:

```
pid_t pgid = getpgrp ( );
```

Jest ono identyczne z następującym wywołaniem:

```
pid_t pgid = getpgid (0);
```

Obie funkcje zwracają identyfikator grupy procesów dla procesu wywołującego. Wywołanie funkcji `getpgrp()` nie może wykonać się niepoprawnie.

Demony

Demon jest procesem, który działa w tle i nie jest podłączony do żadnego terminala sterującego. Demony zwykle uruchamiane są podczas startowania systemu, działają z uprawnieniami administratora lub innego użytkownika specjalnego (takiego jak *apache* lub *postfix*) oraz obsługują zadania na poziomie systemowym. Zgodnie z konwencją nazewniczą nazwy demonów często kończą się na literę *d* (na przykład *crond* i *sshd*), lecz nie jest to wymagane ani powszechnie stosowane.

Nazwa procesu pochodzi od demona Maxwella, którego definicja, będąca wynikiem eksperymentu myślowego, została podana w 1867 roku. Demony są nadnaturalnymi istotami również w mitologii greckiej, istniejącymi gdzieś pomiędzy ludźmi a bogami, posiadającymi dary mocy oraz boskiej wiedzy. W przeciwieństwie do definicji demonów pochodzącej z przekazów i tradycji judeochrześcijańskiej, demony greckie nie musiały być złe. Demony mitologiczne były pomocnikami bogów i wykonywały zadania, których nie chcieli wykonywać mieszkańcy Olimpu — podobnie jak demony uniksowe wykonują zadania, których użytkownicy pierwszoplanowi chcieliby raczej unikać.

Demon posiada dwa wymagania: musi być potomkiem procesu inicjalizującego oraz nie może być podłączony do terminala.

Aby stać się demonem, program musi wykonać następujące czynności:

1. Wywołać funkcję `fork()`. Funkcja ta tworzy nowy proces, który stanie się demonem.
2. Wywołać funkcję `exit()` dla rodzica. Zapewnia to, że jego procesowi rodzicielskiemu („dziadkowi" demona) wystarczy, że proces potomny zakończy się, rodzic demona przestanie istnieć i demon nie będzie liderem grupy procesów. Ostatni warunek wymagany jest w celu poprawnego zakończenia następnej czynności.
3. Wywołać funkcję `setsid()`, tworząc dla demona nową grupę procesów oraz nową sesję, w których będzie on liderem. Dzięki temu terminal sterujący nie będzie związany z procesem (proces tylko stworzy nową sesję, ale nie będzie ona z nim związana).

4. Zmienić katalog roboczy na katalog główny poprzez wywołanie funkcji `chdir()`. Należy to wykonać, ponieważ odziedziczony katalog roboczy może znajdować się gdziekolwiek w systemie plików. Demony działają przez cały czas dostępności systemu i nie można dopuścić, aby jakiś przypadkowy katalog był otwarty, a system plików, w którym ten katalog się znajduje, został odmontowany przez administratora.
5. Zamknąć wszystkie deskryptory plików. Nie należy dziedziczyć otwartych deskryptorów plików i nieświadomie utrzymywać ich w tym stanie.
6. Otworzyć deskryptory plików 0, 1 i 2 (standardowe wejście, wyjście, wyjście błędów) i przeadresować je na */dev/null*.

Poniżej przedstawiono program, który zgodnie z powyższymi regułami przekształca się w demona:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>

int main (void)
{
   pid_t pid;
   int i;

   /* stwórz nowy proces */
   pid = fork ( );
   if (pid == -1)
      return -1;
   else if (pid != 0)
      exit (EXIT_SUCCESS);

   /* stwórz nową sesję i grupę procesów */
   if (setsid ( ) == -1)
      return -1;

   /* ustaw katalog roboczy na katalog główny */
   if (chdir ("/") == -1)
      return -1;

   /* zamknij wszystkie pliki otwarte - użycie opcji NR_OPEN to przesada, lecz działa */
   for (i = 0; i < NR_OPEN; i++)
      close (i);

   /* przeadresuj deskryptory plików 0, 1, 2 na /dev/null */
   open ("/dev/null", O_RDWR); /* stdin */
   dup (0); /* stdout */
   dup (0); /* stderror */

   /* tu należy wykonać czynności demona... */

   return 0;
}
```

Większość systemów uniksowych udostępnia w swoich bibliotekach języka C funkcję `daemon()`, która automatyzuje powyższe działania, zamieniając skomplikowaną procedurę w proste wywołanie:

```
#include <unistd.h>

int daemon (int nochdir, int noclose);
```

Jeśli parametr `nochdir` różny jest od zera, demon nie zmieni swojego katalogu roboczego na katalog główny. Jeśli parametr `noclose` różny jest od zera, demon nie zamknie wszystkich otwartych deskryptorów plików. Parametry te są użyteczne w przypadku, gdy proces rodzicielski już wcześniej ustawił odpowiednie opcje dla demona. Zwykle powinno się jednak przekazywać wartość zero w tych obu parametrach.

W przypadku sukcesu, funkcja zwraca `0`. W przypadku błędu, zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na kod błędu funkcji `fork()` lub `setsid()`.

Zakończenie

W rozdziale tym omówiono podstawy zarządzania procesami w Uniksie, poczynając od tworzenia procesu aż do jego zakończenia. W następnym rozdziale przeanalizowane zostaną bardziej zaawansowane interfejsy zarządzania procesami, jak również funkcje pozwalające na modyfikowanie sposobu szeregowania procesów.

ROZDZIAŁ 6.

Zaawansowane zarządzanie procesami

W rozdziale 5. wprowadzono pojęcie abstrakcji procesu oraz omówiono interfejsy jądra użyte w celu jego tworzenia, kontroli i usuwania. W niniejszym rozdziale informacje te są wykorzystane podczas rozważań na temat *zarządcy procesów* (ang. *scheduler*) i jego algorytmu szeregowania, by zaprezentować zaawansowane interfejsy dla potrzeb zarządzania procesami. Te funkcje systemowe ustalają zachowanie szeregowania oraz semantykę procesu, wpływając na sposób działania zarządcy procesów w dążeniu do celu określonego przez aplikację lub użytkownika.

Szeregowanie procesów

Zarządca procesów jest składnikiem jądra, który decyduje, jaki proces ma być uruchomiony w następnej kolejności. Innymi słowy, zarządca procesów (lub mówiąc prościej: *zarządca*) jest podsystemem jądra, który dzieli skończony zasób czasu procesora pomiędzy procesy systemu. Podczas podejmowania decyzji, jakie procesy i kiedy mają być uruchomione, zarządca jest odpowiedzialny za maksymalizowanie użycia procesora oraz stwarzanie jednocześnie wrażenia, iż wiele procesów jest wykonywanych współbieżnie i płynnie.

W tym rozdziale będzie mowa o procesach uruchamialnych (ang. *runnable*). Proces uruchamialny to taki proces, który przede wszystkim nie jest zablokowany. Procesy współdziałające z użytkownikami, intensywnie zapisujące lub czytające pliki, a także odpowiadające na zdarzenia wejścia i wyjścia lub sieciowe, mają tendencję do zużywania dużej ilości czasu, kiedy są zablokowane podczas oczekiwania na zasoby, które mają stać się dostępne. Nie są one aktywne w czasie tych długich okresów bezczynności (w porównaniu z czasem, który zajmuje wykonanie instrukcji maszynowych). Proces uruchamialny musi również posiadać przynajmniej część swojego *przedziału czasowego* (ang. *timeslice*) — czasu, który przydzielił mu zarządca, aby mógł działać. Jądro umieszcza wszystkie procesy uruchamialne na *liście procesów działających* (ang. *run list*). Gdy tylko proces wykorzysta swój przedział czasowy, jest usuwany z tej listy i nie jest brany pod uwagę jako uruchamialny, do czasu, gdy wszystkie inne procesy uruchamialne wykorzystają swoje przedziały czasowe.

Gdy istnieje tylko jeden proces uruchamialny (lub nie ma ich wcale), działanie zarządcy procesów jest proste. Zarządca udowadnia jednak swoją przydatność w momencie, gdy istnieje więcej procesów uruchamialnych niż procesorów. W takim przypadku niektóre procesy oczywiście muszą działać, gdy inne czekają. Zarządca procesów bierze przede wszystkim odpowiedzialność za decyzję, jaki proces, kiedy i na jak długi czas powinien zostać uruchomiony.

System operacyjny zainstalowany w maszynie posiadającej pojedynczy procesor jest *wielozadaniowy*, gdy potrafi na przemian wykonywać więcej niż jeden proces, stwarzając złudzenie, iż istnieje więcej procesów działających w tym samym czasie. W maszynach wieloprocesorowych wielozadaniowy system operacyjny pozwala procesom faktycznie działać równolegle na różnych procesorach. System operacyjny, który nie jest wielozadaniowy, taki jak DOS, może uruchamiać wyłącznie jeden program w danym momencie.

Wielozadaniowe systemy operacyjne dzielą się na systemy *kooperacyjne* (ang. *cooperative*) oraz systemy *z wywłaszczaniem* (ang. *preemptive*). Linux implementuje drugą formę wielozadaniowości, w której zarządca decyduje, kiedy należy zatrzymać dany proces i uruchomić inny. Wywłaszczenie zwane jest też zawieszeniem działającego procesu. Jeszcze raz powtórzmy — długość czasu, w którym proces działa, zanim zarządca go wywłaszczy, nazywany jest przedziałem czasowym procesu (dosłowne tłumaczenie: „plasterek czasu" — *timeslice* — nazywany tak z powodu przydzielania przez zarządcę każdemu procesowi uruchamialnemu „plasterka" czasu procesora).

W wielozadaniowości kooperatywnej jest odwrotnie — proces nie przestaje działać, dopóki sam nie zadecyduje o swoim zawieszeniu. Czynność procesu, który sam decyduje o przejściu w stan zawieszenia, nazywa się *udostępnianiem czasu procesora* (ang. *yielding*). W sytuacji idealnej procesy często udostępniają czas procesora, lecz system operacyjny jest niezdolny do wymuszenia tego zachowania. Źle działający lub uszkodzony program może być uruchomiony dłużej, niż wynosi zalecany czas, a nawet zawiesić cały system. Z powodu tych problemów, związanych z zagadnieniem wielozadaniowości, nowoczesne systemy operacyjne są generalnie wielozadaniowe z wywłaszczaniem; Linux nie jest tu wyjątkiem.

Zarządca procesów O(1), wprowadzony w wersji jądra 2.5, jest sercem szeregowania procesów w Linuksie[1]. Linuksowy algorytm szeregowania zapewnia wielozadaniowość z wywłaszczeniem razem ze wsparciem dla wielu procesorów, obsługuje *wiązanie procesów do konkretnego procesora* (ang. *processor affinity*), konfiguracje niejednorodnego dostępu do pamięci (ang. *nonuniform memory access*, NUMA), wielowątkowość, procesy czasu rzeczywistego oraz priorytety definiowane przez użytkownika.

Notacja O

Zapis *O(1)* (oznaczający złożoność obliczeniową stałą) jest przykładem notacji O, której używa się w celu określenia złożoności i skalowalności obliczeniowej algorytmu. Formalny zapis wygląda następująco:

$$\text{jeśli } f(x) \text{ posiada złożoność obliczeniową } O(g(x)),$$
$$\text{to}$$
$$\exists c, x' \text{ takie, że } f(x) \le c \cdot g(x), \forall x > x'$$

[1] Dla zainteresowanych: zarządca procesów jest samodzielnym modułem, zdefiniowanym w *kernel/sched.c* w strukturze źródeł jądra.

Tłumacząc to na język polski: złożoność obliczeniowa jakiegoś algorytmu *f* jest zawsze mniejsza lub równa wartości *g* pomnożonej przez pewną odgórnie ustaloną stałą, tak długo, dopóki wartość wejściowa *x* jest większa niż jakaś początkowa wartość *x'*. To znaczy, że *g* jest równe lub większe od *f*; inaczej mówiąc, *g* ogranicza *f* od góry.

Dlatego też *O(1)* zakłada, iż złożoność obliczeniowa badanego algorytmu posiada wartość mniejszą niż pewna stała *c*. Cały ten wywód prowadzi do jednej ważnej obietnicy: linuksowy zarządca procesów będzie zawsze działać z taką samą sprawnością, bez względu na liczbę procesów w systemie. Jest to ważne, ponieważ czynność uruchamiania nowego procesu mogłaby znacząco pociągnąć za sobą przynajmniej jedną, jeśli nie więcej iteracji po liście procesów. W przypadku bardziej prymitywnych zarządców (również tych, którzy istnieli we wcześniejszych wersjach Linuksa) te iteracje mogłyby być powodem powstania potencjalnego wąskiego gardła, gdyby liczba procesów w systemie rosła. W najlepszym razie takie pętle wprowadzają niepewność i brak determinizmu w proces szeregowania.

Przedziały czasowe

Przedział czasowy, który Linux przydziela każdemu procesowi, jest ważną wartością dla ogólnego zachowania i sprawności systemu. Jeśli przedziały czasowe są zbyt duże, procesy muszą oczekiwać przez długi czas pomiędzy uruchomieniami, pogarszając wrażenie równoległego działania. Może to być frustrujące dla użytkownika w przypadku zauważalnych opóźnień. I na odwrót, jeśli przedziały czasowe są zbyt małe, znacząca ilość czasu systemowego jest przeznaczana na przełączanie między aplikacjami i traci się na przykład czasową lokalizację (ang. *temporal locality*) i inne korzyści.

Ustalanie idealnego przedziału czasowego nie jest więc proste. Niektóre systemy operacyjne udzielają procesom dużych przedziałów czasowych, mając nadzieję na zwiększenie przepustowości systemu oraz ogólnej wydajności. Inne systemy operacyjne dają procesom bardzo małe przedziały czasowe w nadziei, iż zapewnią systemowi bardzo dobrą wydajność. Linux wybiera najlepsze elementy z obu możliwych rozwiązań poprzez dynamiczny przydział przedziałów czasowych dla procesów.

Należy zwrócić uwagę, że proces nie musi zużyć całego swojego przedziału czasowego w jednej aktywacji. Proces, któremu przypisano 100 milisekund przedziału czasowego, mógłby działać przez 20 milisekund, a następnie zablokować się na jakimś zasobie, na przykład w oczekiwaniu na dane z klawiatury. Zarządca tymczasowo usunie taki proces z listy procesów działających. Gdy zablokowany zasób stanie się znów dostępny — w tym przypadku, gdy bufor klawiatury przestanie być pusty — zarządca obudzi ten proces. Proces może następnie kontynuować swoje działanie, dopóki nie wykorzysta pozostałych 80 milisekund swojego przedziału czasowego lub nie zablokuje się na jakimś zasobie.

Procesy związane z wejściem i wyjściem a procesy związane z procesorem

Procesy, które w ciągły sposób konsumują wszystkie możliwe przedziały czasowe im przydzielone, są określane mianem procesów *związanych z procesorem* (ang. *processor-bound*). Takie procesy ciągle żądają czasu procesora i będą konsumować wszystko, co zarządca im przydzieli. Najprostszym, trywialnym tego przykładem jest pętla nieskończona. Inne przykłady to obliczenia naukowe, matematyczne oraz przetwarzanie obrazów.

Z drugiej strony, procesy, które przez większość czasu są zablokowane w oczekiwaniu na jakiś zasób, zamiast normalnie działać, nazywane są procesami *związanymi z wejściem i wyjściem* (ang. *I/O-bound*). Procesy związane z wejściem i wyjściem są często wznawiane i oczekują w plikowych operacjach zapisu i odczytu, blokują się podczas oczekiwania na dane z klawiatury lub czekają na akcję użytkownika polegającą na ruchu myszką. Przykłady aplikacji związanych z wejściem i wyjściem to programy użytkowe, które robią niewiele poza tym, że generują wywołania systemowe żądające od jądra, aby wykonał operacje wejścia i wyjścia, takie jak `cp` lub `mv`, a także wiele aplikacji GUI, które zużywają dużo czasu oczekując na akcję użytkownika.

Aplikacje związane z procesorem oraz aplikacje związane z wejściem i wyjściem różnią się między sobą sposobem działania zarządcy, który stara się nadać im odpowiednie przywileje. Aplikacje związane z procesorem wymagają możliwie największych przedziałów czasowych, pozwalających im na poprawienie współczynnika używania pamięci podręcznej (poprzez czasową lokalizację) oraz jak najszybciej kończą swoje działanie. W przeciwieństwie do nich procesy związane z wejściem i wyjściem nie wymagają koniecznie dużych przedziałów czasowych, ponieważ standardowo działają tylko przez krótki czas przed wysłaniem żądań związanych z wejściem i wyjściem oraz blokowaniem się na jakimś zasobie z jądra systemu. Procesy związane z wejściem i wyjściem czerpią jednak korzyści z tego, iż zarządca przez cały czas je obsługuje. Im szybciej jakaś aplikacja może ponownie uruchomić się po zablokowaniu się i wysłaniu większej liczby żądań wejścia i wyjścia, tym lepiej potrafi ona używać urządzeń systemowych. Co więcej, im szybciej aplikacja czekająca na akcję użytkownika zostanie zaszeregowana, tym bardziej sprawia ona wrażenie płynnego działania dla tego użytkownika.

Dopasowywanie potrzeb procesów związanych z procesorem oraz wejściem i wyjściem nie jest łatwe. Zarządca w Linuksie przystępuje do rozpoznania i dostarczania zalecanego sposobu traktowania aplikacji związanych z wejściem i wyjściem: aplikacje intensywnie wykonujące operacje wejścia i wyjścia otrzymują zwiększony priorytet, podczas gdy aplikacje silnie związanie z procesorem są „karane" poprzez obniżenie ich priorytetu.

W rzeczywistości większość aplikacji stanowi połączenie procesów związanych z procesorem oraz z wejściem i wyjściem. Kodowanie oraz dekodowanie strumieni dźwięku i obrazu jest dobrym przykładem typu aplikacji, który opiera się jednoznacznej kwalifikacji. Wiele gier to również aplikacje o typie mieszanym. Nie zawsze jest możliwa identyfikacja skłonności danej aplikacji; w określonym momencie dany proces może zachowywać się w różny sposób.

Szeregowanie z wywłaszczaniem

Gdy dany proces wykorzysta swój przedział czasowy, jądro zawiesza go, a rozpoczyna wykonywanie nowego procesu. Jeśli w systemie nie istnieje więcej procesów uruchamialnych, jądro pobiera grupę procesów z wykorzystanymi przedziałami czasowymi, uzupełnia te przedziały oraz uruchamia procesy ponownie. W ten sposób wszystkie procesy ostatecznie są uruchamiane, nawet jeśli w systemie istnieją procesy o wyższym priorytecie; procesy niskopriorytetowe muszą czekać, aż procesy o wysokim priorytecie wyczerpią swoje przedziały czasowe lub zablokują się. To zachowanie formułuje ważną, lecz ukrytą regułę szeregowania w systemach Unix: wszystkie procesy muszą kontynuować swoje działanie.

Jeśli w systemie nie ma już uruchamialnych procesów, jądro „uruchamia" *proces jałowy* (ang. *idle process*). Proces jałowy tak naprawdę nie jest w ogóle procesem, w rzeczywistości również

nie działa (co za ulga dla baterii). Jest natomiast specjalnym podprogramem, uruchamianym przez jądro, aby uprościć algorytm szeregowania oraz ułatwić rozliczanie zadań. Czas jałowy to po prostu czas zużywany na proces jałowy.

Jeśli proces działa, a proces o wyższym priorytecie staje się uruchamialny (być może dlatego, że został zablokowany podczas oczekiwania na naciśnięcie klawisza, a użytkownik właśnie wprowadził tekst z klawiatury), aktualnie działający proces zostaje natychmiast zawieszony, a jądro uruchamia proces o wyższym priorytecie. Tak więc nigdy nie ma procesów o wyższym priorytecie niż aktualnie działający proces, które są uruchamialne, ale nie działają. Proces działający jest zawsze procesem uruchamialnym o najwyższym priorytecie w systemie.

Wątki

Wątki są to jednostki wykonawcze w obrębie jednego procesu. Wszystkie procesy mają przynajmniej jeden wątek. Każdy wątek posiada własną wirtualizację procesora: swój zestaw rejestrów, wskaźnik instrukcji oraz stan procesora. Podczas gdy większość procesów posiada wyłącznie jeden wątek, niektóre procesy mogą mieć większą liczbę wątków, wykonujących różne zadania, lecz dzielących tę samą przestrzeń adresową (i taką samą pamięć dynamiczną, pliki mapowane, kod itd.), listę otwartych plików oraz inne zasoby jądra.

Jądro Linuksa posiada interesujący i unikalny widok wątków. Zasadniczo jądra systemów nie posiadają takiej możliwości. Dla jądra linuksowego wszystkie wątki są oddzielnymi procesami. Ogólnie rzecz biorąc, nie ma różnicy pomiędzy dwoma niepowiązanymi ze sobą procesami a dwoma wątkami wewnątrz pojedynczego procesu. Jądro po prostu widzi wątki jako procesy, które dzielą te same zasoby. To znaczy, że jądro traktuje proces składający się z dwóch wątków jako dwa rozdzielne procesy, które dzielą zbiór zasobów jądra (przestrzeń adresową, listę otwartych plików itd.).

Programowanie wielowątkowe to sztuka programowania przy użyciu wątków. Najbardziej popularnym API linuksowym służącym programowaniu z wykorzystaniem wątków jest API ustandaryzowane przez *IEEE Std 1000.1c-1995* (*POSIX 1995* lub *POSIX.1c*). Taka biblioteka, która implementuje API dla wątków POSIX, nazywana jest często przez programistów *pthreads*. Omówienie biblioteki *pthreads* znajduje się poza zakresem tematów, które porusza ta książka. Zamiast tego szczegółowej analizie zostaną poddane interfejsy, na których ta biblioteka jest zbudowana.

Udostępnianie czasu procesora

Chociaż Linux jest wielozadaniowym systemem operacyjnym z wywłaszczaniem, dostarcza również funkcję systemową, która pozwala procesom jawnie zawieszać działanie i informować zarządcę, by wybrał nowy proces do uruchomienia:

```
#include <sched.h>
int sched_yield (void);
```

Wywołanie funkcji `sched_yield()` skutkuje zawieszeniem aktualnie działającego procesu, po czym zarządca procesów wybiera nowy proces, aby go uruchomić, w taki sam sposób, jakby jądro samo wywłaszczyło aktualny proces w celu uruchomienia nowego. Warte uwagi jest to, że jeśli nie istnieje żaden inny proces uruchamialny, co jest częstym przypadkiem, wówczas

proces udostępniający natychmiast ponowi swoje działanie. Z powodu przypadkowości połączonej z ogólnym przekonaniem, iż zazwyczaj istnieją lepsze metody, użycie tego wywołania systemowego nie jest popularne.

W przypadku sukcesu funkcja zwraca `0`; w przypadku porażki zwraca `-1` oraz ustawia `errno` na odpowiednią wartość kodu błędu. W Linuksie oraz — bardziej niż prawdopodobnie — w większości systemów Unix wywołanie funkcji `sched_yield()` nie może się nie udać i dlatego też zawsze zwraca wartość zero. Drobiazgowy programista może jednak ciągle próbować sprawdzać zwracaną wartość:

```
if (sched_yield ( ))
   perror ("sched_yield");
```

Prawidłowe sposoby użycia sched_yield()

W praktyce istnieje kilka (jeśli w ogóle istnieją) prawidłowych sposobów użycia funkcji `sched_yield()` w poprawnym wielozadaniowym systemie z wywłaszczeniem, jak na przykład w Linuksie. Jądro jest w pełni zdolne do podjęcia optymalnych oraz najbardziej efektywnych decyzji dotyczących szeregowania — oczywiście, jądro jest lepiej niż sama aplikacja wyposażone w mechanizmy, które pozwalają decydować, co i kiedy należy wywłaszczyć. Oto, dlaczego w systemach operacyjnych odchodzi się od wielozadaniowości kooperatywnej na rzecz wielozadaniowości z wywłaszczeniem.

Dlaczego więc w ogóle istnieje funkcja systemowa „Przeszereguj mnie"? Odpowiedź można znaleźć w aplikacjach oczekujących na zdarzenia zewnętrzne, które mogą być spowodowane przez użytkownika, element sprzętowy albo inny proces. Na przykład, jeśli jeden proces musi czekać na inny, pierwszym narzucającym się rozwiązaniem tego problemu jest: „po prostu udostępnij czas procesora, dopóki inny proces się nie zakończy". Jako przykład mogłaby wystąpić implementacja prostego konsumenta w parze producent-konsument, która byłaby podobna do następującego kodu:

```
/* konsument... */
do
{
   while (producer_not_ready ())
      sched_yield ();
   process_data ();
} while (!time_to_quit());
```

Na szczęście programiści systemu Unix nie są skłonni do tworzenia kodu tego typu. Programy uniksowe są zwykle sterowane zdarzeniami i zamiast stosować funkcję `sched_yield()` dążą do używania pewnego rodzaju mechanizmu blokowania (takiego jak potok — ang. *pipe*) pomiędzy konsumentem a producentem. W tym przypadku konsument próbuje czytać z potoku, będąc jednak zablokowanym, dopóki nie pojawią się dane. Z drugiej strony, producent zapisuje do potoku, gdy tylko dostępne stają się nowe dane. Przenosi to odpowiedzialność za koordynację z procesu należącego do poziomu użytkownika, na przykład z pętli zajmujących czas procesora do jądra, które może optymalnie zarządzać taką sytuacją przez przełączanie procesów w stan uśpienia oraz budzenie ich tylko w przypadku potrzeby. Ogólnie rzecz biorąc, programy uniksowe powinny dążyć do rozwiązań sterowanych zdarzeniami, które opierają się na blokowanych deskryptorach plików.

Jedna sytuacja do niedawna koniecznie wymagała `sched_yield()` — blokowanie wątku z poziomu użytkownika. Gdy jakiś wątek próbował uzyskać blokadę, która jest już w posiadaniu

innego wątku, wówczas udostępniał on czas procesora dopóty, dopóki blokada nie została zwolniona. Bez wsparcia blokad z poziomu użytkownika ze strony jądra to podejście było najprostsze i najbardziej efektywne. Na szczęście nowoczesna implementacja wątków w Linuksie (*New POSIX Threading Library* — NPTL) wprowadziła optymalne rozwiązanie przy użyciu mechanizmu *futex*, który dostarcza procesorowi wsparcia dla systemu blokowania z poziomu użytkownika.

Jeszcze innym przykładem użycia funkcji `sched_yield()` jest zagadnienie „sympatycznego działania": program intensywnie obciążający procesor może wywoływać co jakiś czas `sched_yield()`, przez co minimalizuje swoje obciążenie w systemie. Rozwiązanie to ma być może szlachetny cel, lecz niestety posiada dwie wady. Po pierwsze, jądro potrafi podejmować globalne decyzje dotyczące szeregowania dużo lepiej niż indywidualne procesy i w konsekwencji odpowiedzialność za zapewnienie płynności operacji w systemie powinna spoczywać na zarządcy procesów, a nie na samych procesach. Reasumując: premiowanie interakcyjności przez zarządcę jest ukierunkowane w celu nagradzania aplikacji korzystających intensywnie z portów wejścia i wyjścia, natomiast karania tych, które obciążają procesor. Po drugie, za zmniejszenie obciążenia aplikacji korzystającej intensywnie z procesora z jednoczesnym wyróżnieniem innych programów odpowiada użytkownik, a nie pojedyncza aplikacja. Użytkownik może zmodyfikować swoje preferencje dotyczące wydajności aplikacji poprzez użycie komendy powłoki systemowej `nice`, która będzie omawiana w dalszej części tego rozdziału.

Udostępnianie czasu procesora — kiedyś i teraz

Zanim wprowadzone zostało jądro Linuksa w wersji 2.6, wywołanie funkcji `sched_yield()` powodowało tylko mało ważny skutek. Jeśli był dostępny inny proces uruchamialny, jądro mogło przełączyć się na niego i umieścić proces wywołujący na końcu listy procesów uruchamialnych. Krótko mówiąc, jądro mogło przeszeregować proces wywołujący. W prawdopodobnym przypadku, gdyby nie istniał inny proces uruchamialny, proces wywołujący po prostu kontynuowałby swoje działanie.

W jądrze w wersji 2.6 zachowanie to uległo zmianie. Aktualny algorytm wygląda następująco:

1. Czy proces jest procesem czasu rzeczywistego? Jeśli tak, należy dołączyć go na koniec listy procesów uruchamialnych i na tym zakończyć (jest to stare zachowanie). Jeśli nie, należy przejść do następnego punktu (aby dowiedzieć się więcej na temat procesów czasu rzeczywistego, należy przeczytać podrozdział zatytułowany Systemy czasu rzeczywistego, znajdujący się w dalszej części tego rozdziału).
2. Należy zupełnie usunąć ten proces z listy procesów uruchamialnych i umieścić go na liście procesów przedawnionych. Powoduje to, że wszystkie procesy uruchamialne muszą się wykonać i zużyć swoje przedziały czasowe, zanim proces wywołujący razem z innymi przedawnionymi procesami jest zdolny do kontynuowania swojego działania.
3. Należy zaszeregować następny proces uruchamialny z listy procesów przeznaczonych do działania.

Sumaryczny efekt wywołania funkcji `sched_yield()` jest więc taki sam jak w przypadku, gdyby proces wykorzystał swój przedział czasowy. Zachowanie to różni się od zachowania istniejącego dla wcześniejszych wersji jądra, gdzie skutek wywołania `sched_yield()` był łagodniejszy (równoznaczny z następującym sformułowaniem: „jeśli inny proces jest gotowy i oczekuje, uruchom go na chwilę, ale wróć zaraz z powrotem do mnie").

Jednym powodem dla przeprowadzenia tej modyfikacji była ochrona przed nienormalnym przypadkiem tak zwanego „ping-ponga". Załóżmy, że istnieją dwa procesy: A i B, oba wywołujące funkcję `sched_yield()`. Załóżmy też, że są to jedyne uruchamialne procesy (mogą istnieć również inne procesy zdolne do działania, lecz żaden z nich nie posiada niezerowych przedziałów czasowych). W przypadku poprzedniego zachowania funkcji `sched_yield()` wynikiem tej sytuacji jest to, iż jądro na zmianę próbuje szeregować oba procesy, każdemu z nich jednak mówiąc: „Nie, przeszereguję kogoś innego!". Taki stan rzeczy trwa do momentu, gdy oba procesy wyczerpią swoje przedziały czasowe. Gdyby narysowano wykres przedstawiający wybór procesów przez zarządcę, wyglądałby on tak: „A, B, A, B, A, B" i tak dalej — stąd przydomek „ping-pong".

Nowy algorytm `sched_yield()` nie dopuszcza do takiej sytuacji. Gdy tylko proces A chce udostępnić czas procesora, zarządca usuwa go z listy procesów uruchamialnych. Podobnie gdy proces B wysyła to samo żądanie, zarządca również usuwa go z listy procesów uruchamialnych. Zarządca nie będzie próbował uruchamiać procesu A lub B, dopóki nie będzie już żadnych procesów uruchamialnych, zapobiegając efektowi „ping-ponga" i pozwalając innym procesom otrzymać właściwy przydział czasu procesora.

Zgodnie z powyższym, gdy pojawia się prośba o udostępnienie czasu procesora, proces powinien rzeczywiście chcieć go udostępnić!

Priorytety procesu

Dyskusja w tym podrozdziale dotyczy zwykłych procesów, niebędących procesami czasu rzeczywistego. Procesy czasu rzeczywistego wymagają odmiennych kryteriów szeregowania oraz oddzielnego systemu priorytetów. Będzie to omówione w dalszej części rozdziału.

Linux nie szereguje procesów w sposób przypadkowy. Zamiast tego aplikacjom przypisywane są *priorytety*, które wpływają na to, kiedy i jak długo ich odpowiednie procesy są uruchamiane. Od początku swojego istnienia Unix nazywał te priorytety *poziomami uprzejmości* (ang. *nice values*), ponieważ ideą ukrytą za tą nazwą było to, aby „być uprzejmym" dla innych procesów znajdujących się w systemie poprzez obniżanie priorytetu aktualnego procesu, zezwalając przez to innym procesom na konsumowanie większej ilości czasu procesora.

Poziom uprzejmości ustala dla procesu czas jego uruchomienia. Linux szereguje procesy uruchamialne w kolejności od najwyższego do najniższego priorytetu: proces o wyższym priorytecie jest uruchomiony przed procesem o niższym priorytecie. Poziom uprzejmości ustanawia również rozmiar przedziału czasowego dla procesu.

Poprawne poziomy uprzejmości zawierają się w granicach od –20 do 19 włącznie, z domyślną wartością równą zeru. W pewien sposób dezorientujące jest to, iż im niższy poziom uprzejmości dla danego procesu, tym wyższy jego priorytet oraz większy przedział czasu; odwrotnie, im wyższa wartość, tym niższy priorytet procesu i mniejszy przedział czasu. Dlatego też zwiększanie poziomu uprzejmości dla procesu jest „uprzejme" dla reszty systemu. Odwrócenie wartości liczbowych wprawia niestety w zakłopotanie. Gdy mówi się, że proces ma „wyższy priorytet", oznacza to, że częściej zostaje wybrany do uruchomienia i może działać dłużej niż procesy niskopriorytetowe. Taki proces posiada jednak niższy poziom uprzejmości.

nice()

Linux dostarcza kilka funkcji systemowych w celu odczytania oraz ustawienia poziomu uprzejmości dla procesu. Najprostszą z nich jest funkcja `nice()`:

```
#include <unistd.h>
int nice (int inc);
```

Udane wywołanie funkcji `nice()` zwiększa poziom uprzejmości procesu o liczbę przekazaną przez parametr `inc` oraz zwraca uaktualnioną wartość. Tylko proces z uprawnieniami `CAP_SYS_NICE` (w rzeczywistości proces, którego właścicielem jest administrator) może przekazywać ujemną wartość w parametrze `inc`, zmniejszając swój poziom uprzejmości i przez to zwiększając swój priorytet. Zgodnie z tym procesy niebędące procesami należącymi do administratora mogą jedynie obniżać swój priorytet (poprzez zwiększanie swojego poziomu uprzejmości).

W przypadku błędu wykonania `nice()` zwraca wartość `-1`. Ponieważ jednak `nice()` zwraca nową wartość poziomu uprzejmości, `-1` jest równocześnie poprawną wartością zwrotną. Aby odróżnić poprawne wywołanie od błędnego, można wyzerować wartość `errno` przed wykonaniem wywołania, a następnie ją sprawdzić. Na przykład:

```
int ret;
errno = 0;
ret = nice (10); /* zwiększ nasz poziom uprzejmości o 10 */
if (ret == -1 && errno != 0)
   perror ("nice");
else
   printf ("Poziom uprzejmości wynosi aktualnie %d\n", ret);
```

Linux zwraca tylko jeden kod błędu: `EPERM`, informując, iż proces wywołujący spróbował zwiększyć swój priorytet (poprzez ujemną wartość parametru `inc`), lecz nie posiadał uprawnienia `CAP_SYS_NICE`. Inne systemy zwracają również kod błędu `EINVAL`, gdy `inc` próbuje ustalić nową wartość poziomu uprzejmości poza dopuszczalnym zakresem, lecz Linux tego nie robi. Zamiast tego Linux zaokrągla niewłaściwe wartości `inc` w razie potrzeby w górę lub w dół do wielkości będącej odpowiednią granicą dopuszczalnego zakresu.

Przekazanie zera w parametrze `inc` jest prostym sposobem na uzyskanie aktualnej wartości poziomu uprzejmości:

```
printf ("Poziom uprzejmości wynosi obecnie %d\n", nice (0));
```

Często proces chce ustawić bezwzględną wartość poziomu uprzejmości, zamiast przekazywać względną wielkość różnicy. Można to uzyskać poprzez wykonanie poniższego kodu:

```
int ret, val;
/* pobierz aktualną wartość poziomu uprzejmości */
val = nice (0);
/* chcemy ustawić poziom uprzejmości na 10 */
val = 10 - val;
errno = 0;
ret = nice (val);
if (ret == -1 && errno != 0)
   perror ("nice");
else
   printf ("Poziom uprzejmości wynosi aktualnie %d\n", ret);
```

getpriority() i setpriority()

Preferowanym rozwiązaniem jest użycie funkcji systemowych `getpriority()` oraz `setprio ↪rity()`, które udostępniają więcej możliwości kontroli, lecz są bardziej złożone w działaniu:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

Funkcje te działają na procesie, grupie procesów lub użytkowniku, co odpowiednio ustalają parametry `which` oraz `who`. Parametr `which` musi posiadać jedną z następujących wartości: `PRIO_PROCESS`, `PRIO_PGRP` lub `PRIO_USER`; w tym przypadku parametr `who` określa odpowiednio identyfikator procesu, identyfikator grupy procesów lub identyfikator użytkownika.

Wywołanie funkcji `getpriority()` zwraca najwyższy priorytet (najniższą liczbową wartość poziomu uprzejmości) dla każdego z podanych procesów. Wywołanie funkcji `setpriority()` ustawia priorytet wszystkich podanych procesów na wartość `prio`. Podobnie jak w przypadku funkcji `nice()`, tylko proces posiadający uprawnienia `CAP_SYS_NICE` może zwiększyć priorytet procesu (zmniejszyć liczbową wartość poziomu uprzejmości). Co więcej, tylko proces z tym uprawnieniem może zwiększyć lub zmniejszyć priorytet procesu niebędącego w posiadaniu przez wywołującego go użytkownika.

Podobnie jak `nice()`, również `getpriority()` zwraca `-1` w przypadku błędu. Ponieważ jest to również poprawna wartość zwrotna, programiści powinni zerować `errno` przed wywołaniem funkcji, jeśli zamierzają obsługiwać przypadki błędów. Wywołanie `setpriority()` nie powoduje takich problemów —`setpriority()` zawsze zwraca `0` w przypadku powodzenia, natomiast `-1` w przypadku niepowodzenia.

Następujący kod zwraca aktualną wartość priorytetu procesu:

```
int ret;
ret = getpriority (PRIO_PROCESS, 0);
printf ("Poziom uprzejmości wynosi %d\n", ret);
```

Poniższy kod ustawia priorytet dla wszystkich procesów z aktualnej grupy procesów na wartość 10:

```
int ret;
ret = setpriority (PRIO_PGRP, 0, 10);
if (ret == -1)
   perror ("setpriority");
```

W przypadku błędu obie funkcje ustawiają `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces zamierzał zwiększyć priorytet, lecz nie posiada uprawnienia `CAP_SYS_NICE` (tylko dla `setpriority()`).

`EINVAL`
: Wartość przekazana przez parametr `which` nie była jedną z następujących: `PRIO_PROCESS`, `PRIO_PGRP` lub `PRIO_USER`.

`EPERM`
: Efektywny identyfikator użytkownika procesu branego pod uwagę nie zgadza się z efektywnym identyfikatorem użytkownika procesu działającego; proces działający dodatkowo nie posiada uprawnienia `CAP_SYS_NICE` (tylko dla `setpriority()`).

ESRCH
: Nie znaleziono procesu spełniającego wymagań, które zostały określone przez parametry `which` oraz `who`.

Priorytety wejścia i wyjścia

W ramach dodatku do priorytetów szeregowania Linux pozwala procesom określić *priorytety wejścia i wyjścia*. Ta wartość wpływa na względny priorytet żądań wejścia i wyjścia dla procesów. Zarządca wejścia i wyjścia z jądra systemu (omówiony w rozdziale 4.) obsługuje żądania pochodzące z procesów o wyższym priorytecie wejścia i wyjścia przed żądaniami z procesów o niższym priorytecie wejścia i wyjścia.

Zarządcy wejścia i wyjścia domyślnie wykorzystują poziom uprzejmości procesu, aby ustalić priorytet wejścia i wyjścia. Dlatego też ustawienie poziomu uprzejmości automatycznie zmienia priorytet wejścia i wyjścia. Jądro Linuksa dostarcza jednakże dodatkowo dwóch funkcji systemowych, w celu jawnego ustawienia i uzyskania informacji dotyczących priorytetu wejścia i wyjścia, niezależnie od wartości poziomu uprzejmości:

```
int ioprio_get (int which, int who)
int ioprio_set (int which, int who, int ioprio)
```

Niestety jądro nie udostępnia jeszcze tych funkcji systemowych, a *glibc* nie umożliwia żadnego dostępu do nich z poziomu użytkownika. Bez wsparcia przez *glibc* używanie ich jest w najlepszym przypadku niewygodne. Co więcej, w momencie, gdy biblioteka *glibc* zacznie udzielać wsparcia, może zdarzyć się, iż jej interfejs nie będzie kompatybilny z tymi funkcjami. Dopóki nie ma takiego wsparcia, istnieją dwa przenośne sposoby pozwalające modyfikować priorytety wejścia i wyjścia: poprzez poziomy uprzejmości lub program użytkowy o nazwie *ionice*, który jest składnikiem pakietu *util-linux*[2].

Nie wszyscy zarządcy wejścia i wyjścia wspierają priorytety wejścia i wyjścia. Wyjątkiem jest zarządca wejścia i wyjścia o nazwie *Complete Fair Queuing* (CFQ); inni typowi zarządcy aktualnie nie posiadają wsparcia dla priorytetów wejścia i wyjścia. W przypadku, gdy dany zarządca wejścia i wyjścia nie wspiera priorytetów wejścia i wyjścia, są one przez niego milcząco pomijane.

Wiązanie procesów do konkretnego procesora

Linux wspiera wieloprocesorowość dla pojedynczego systemu. Poza procesem ładowania systemu większość pracy niezbędnej dla zapewnienia poprawnego działania systemu wieloprocesorowego zrzucona jest na zarządcę procesów. Na maszynie z wieloprocesorowością symetryczną (SMP) zarządca procesów musi decydować, który proces powinien uruchamiać na każdym procesorze. Z tej odpowiedzialności wynikają dwa wyzwania: zarządca musi dążyć do tego, aby w pełni wykorzystywać wszystkie procesory w systemie, ponieważ sytuacja, gdy procesor jest nieobciążony, a proces oczekuje na uruchomienie, prowadzi do nieefektywnego działania.

Gdy jednak proces został zaszeregowany dla określonego procesora, zarządca procesów powinien dążyć, aby szeregować go do tego samego procesora również w przyszłości. Jest to korzystne, gdyż *migracja* procesu z jednego procesora do innego pociąga za sobą pewne koszty.

[2] Pakiet *util-linux* jest osiągalny pod adresem *http://www.kernel.org/pub/linux/utils/util-linux*. Jest on licencjonowany zgodnie z Powszechną Licencją Publiczną GNU.

Największa część tych kosztów dotyczy *skutków buforowania* (ang. *cache effects*) podczas migracji. Z powodu określonych konstrukcji nowoczesnych systemów SMP pamięci podręczne przyłączone do każdego procesora są niezależne i oddzielone od siebie. Oznacza to, że dane z jednej pamięci podręcznej nie powielają się w innej. Dlatego też, jeśli proces przenosi się do nowego procesora i zapisuje nowe informacje w pamięci, dane w pamięci podręcznej poprzedniego procesora mogą stracić aktualność. Poleganie na tej pamięci podręcznej może wówczas spowodować uszkodzenie danych. Aby temu przeciwdziałać, pamięci podręczne unieważniają każde inne dane, ilekroć buforują nowy fragment pamięci. Zgodnie z tym określony fragment danych znajduje się dokładnie tylko w jednej pamięci podręcznej procesora w danym momencie (zakładając, że dane są w ogóle buforowane). Gdy proces przenosi się z jednego procesora do innego, pociąga to za sobą następujące koszty: dane buforowane nie są już dostępne dla procesu, który się przeniósł, a dane w źródłowym procesorze muszą być unieważnione. Z powodu tych kosztów zarządca procesów próbuje utrzymać proces na określonym procesorze tak długo, jak to jest możliwe.

Dwa cele zarządcy procesów są oczywiście potencjalnie sprzeczne ze sobą. Jeśli jeden procesor jest znacząco bardziej obciążony niż drugi albo, co gorsza, jeśli jeden procesor jest całkowicie zajęty, podczas gdy drugi jest bezczynny, wówczas zaczyna mieć sens przeszeregowanie niektórych procesów do procesora mniej obciążonego. Decyzja, kiedy należy przenieść procesy w przypadku takiego braku równowagi, zwana *wyrównywaniem obciążenia*, odgrywa bardzo ważną rolę w wydajności maszyn SMP.

Wiązanie procesów dla konkretnego procesora odnosi się do prawdopodobieństwa, że proces zostanie konsekwentnie zaszeregowany do tego samego procesora. Termin *miękkie wiązanie* (ang. *soft affinity*) określa naturalną skłonność zarządcy, aby kontynuować szeregowanie procesu na tym samym procesorze. Jak już powiedziano, jest to wartościowa cecha. Zarządca linuksowy szereguje te same procesy na tych samych procesorach przez tak długi czas, jak to jest możliwe, przenosząc proces z jednego procesora na inny jedynie w przypadku wyjątkowego braku równowagi. Pozwala to zarządcy zmniejszać skutki buforowania w przypadku migracji procesów i zapewnia, że wszystkie procesory w systemie są równo obciążone.

Czasami jednak użytkownik lub aplikacja chcą wymusić powiązanie proces-procesor. Dzieje się tak często, gdy sam proces jest wrażliwy na buforowanie i wymaga pozostawienia go na tym samym procesorze. Wiązanie procesu do konkretnego procesora, które zostaje wymuszone przez jądro systemu, nazywa się *twardym wiązaniem* (ang. *hard affinity*).

sched_getaffinity() i sched_setaffinity()

Procesy dziedziczą wiązania do procesora od swych rodziców, a także — domyślnie — mogą działać na dowolnym procesorze. Linux udostępnia dwie funkcje systemowe pozwalające na odczytanie oraz ustawienie twardego wiązania dla procesu:

```
#define _GNU_SOURCE
#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
```

```
void CPU_ZERO (cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize, const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize, cpu_set_t *set);
```

Wywołanie funkcji `sched_getaffinity()` odczytuje wiązanie do procesora dla procesu o identyfikatorze `pid` i zapisuje go w zmiennej specjalnego typu `cpu_set_t`, która jest dostępna poprzez specyficzne makra. Jeśli `pid` jest równe zeru, funkcja zwraca wiązanie dla aktualnego procesu. Parametr `setsize` określa rozmiar typu `cpu_set_t`, który może być używany przez *glibc* dla potrzeb kompatybilności z przyszłymi modyfikacjami dotyczącymi rozmiaru tego typu. W przypadku sukcesu `sched_getaffinity()` zwraca `0`; w przypadku błędu zwraca `−1` oraz ustawia `errno`. Oto przykład:

```
cpu_set_t set;
int ret, i;
CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
   perror ("sched_getaffinity");
for (i = 0; i < CPU_SETSIZE; i++)
{
   int cpu;
   cpu = CPU_ISSET (i, &set);
   printf ("procesor = %i jest %s\n", i, cpu ? "ustawiony" : "wyzerowany");
}
```

Na początku kodu następuje użycie makra `CPU_ZERO`, by wyzerować wszystkie bity w zmiennej `set`. W dalszej części przeprowadzana jest iteracja w pętli po wszystkich elementach tej zmiennej. Ważne jest to, iż `CPU_SETSIZE` nie jest rozmiarem zmiennej `set` (*nigdy* nie należy używać tej wartości w `setsize`!), lecz liczbą procesorów, które mogłyby być potencjalnie reprezentowane przez `set`. Ponieważ aktualna implementacja definiuje przedstawienie każdego procesora jako jeden bit, `CPU_SETSIZE` jest dużo większe niż `sizeof(cpu_set_t)`. `CPU_ISSET` jest używane, aby sprawdzić, czy dany procesor o numerze `i` jest związany (albo nie) z danym procesem. Makro zwraca zero, gdy nie jest związany, lub wartość niezerową, gdy jest związany.

Tylko procesory zainstalowane fizycznie w systemie są rozpoznawane jako związane. Dlatego też powyższy fragment kodu uruchomiony w systemie z dwoma procesorami zwróci następujący wynik:

```
procesor = 0 jest ustawiony
procesor = 1 jest ustawiony
procesor = 2 jest wyzerowany
procesor = 3 jest wyzerowany
...
procesor = 1023 jest wyzerowany
```

Jak wynika z rezultatów zadziałania kodu, wartość `CPU_SETSIZE` (o indeksie rozpoczynającym się od zera) wynosi aktualnie 1024.

W przykładzie mieliśmy do czynienia wyłącznie z procesorem nr 0 i 1, ponieważ są to jedyne obecne fizycznie procesory w systemie. Być może zaistnieje potrzeba, aby zapewnić konfigurację, w której dany proces działa wyłącznie na procesorze o numerze 0, a w ogóle nie działa na procesorze o numerze 1. To zadanie realizuje poniższy kod:

```
cpu_set_t set;
int ret, i;
CPU_ZERO (&set); /* wyzeruj struktury dla wszystkich procesorów */
CPU_SET (0, &set); /* zezwól na procesor 0 */
```

```
CPU_CLR (1, &set); /* zablokuj procesor 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
   perror ("sched_setaffinity");
for (i = 0; i < CPU_SETSIZE; i++)
{
   int cpu;
   cpu = CPU_ISSET (i, &set);
   printf ("procesor = %i jest %s\n", i, cpu ? "ustawiony" : "wyzerowany");
}
```

Program rozpoczyna się, jak zawsze, od zerowania zmiennej `set` za pomocą makra `CPU_ZERO`. Następnie ustawia się procesor 0 przy użyciu `CPU_SET` oraz zeruje procesor 1 przy użyciu `CPU_CLR`. Operacja `CPU_CLR` jest nadmiarowa, gdyż wcześniej już wyzerowano całą strukturę `set`; jest ona jednak użyta, aby zapewnić kompletność kodu.

Uruchomienie tego programu na tym samym dwuprocesorowym systemie zakończy się trochę innymi niż poprzednio wynikami:

```
procesor = 0 jest ustawiony
procesor = 1 jest wyzerowany
procesor = 2 jest wyzerowany
...
procesor = 1023 jest wyzerowany
```

Obecnie procesor 1 jest wyzerowany. Proces będzie działać wyłącznie na procesorze 0, bez względu na wszystko!

Możliwe są cztery wartości `errno`:

`EFAULT`
: Dostarczony wskaźnik znajdował się poza przestrzenią adresową procesu lub był ogólnie nieprawidłowy.

`EINVAL`
: W tym przypadku w systemie nie istniały fizycznie procesory, które zostały odblokowane w zmiennej `set` (tylko dla `sched_setaffinity()`) lub też `setsize` było mniejsze niż rozmiar wewnętrznej struktury danych jądra użytej w celu reprezentacji zbioru procesorów.

`EPERM`
: Proces reprezentowany przez `pid` nie zgadza się z aktualnym efektywnym identyfikatorem użytkownika procesu wywołującego; proces dodatkowo nie posiada uprawnienia `CAP_SYS_NICE`.

`ESRCH`
: Nie znaleziono procesu posiadającego identyfikator `pid`.

Systemy czasu rzeczywistego

W systemach komputerowych termin *czas rzeczywisty* (ang. *real-time*) jest często powodem pewnego zamieszania i niezrozumienia. System jest systemem czasu rzeczywistego, jeśli jego parametry mieszczą się w zakresie *granicznych parametrów operacyjnych* (ang. *operational deadlines*): czasach reakcji — minimalnym oraz narzuconym — pomiędzy zdarzeniem i odpowiedzią na to zdarzenie. Znanym systemem czasu rzeczywistego jest ABS (ang. *antilock braking system*), spotykany obecnie w prawie wszystkich nowoczesnych samochodach. W tym systemie, jeśli nastąpi rozpoczęcie procesu hamowania, komputer zaczyna odpowiednio regulować docisk

hamulców, impulsowo zwiększając go do maksimum i następnie zmniejszając wiele razy na sekundę. Chroni to koła przed zablokowaniem, które mogłoby spowodować pogorszenie warunków hamowania lub nawet wprowadzić samochód w niekontrolowany poślizg. W takim układzie granicznymi parametrami operacyjnymi są: szybkość reakcji systemu na stan zablokowania koła oraz szybkość uzyskania przez system określonego nacisku w hamulcach.

Większość nowoczesnych systemów operacyjnych, także Linux, zapewnia pewien poziom wsparcia dla operacji czasu rzeczywistego.

Systemy ścisłego oraz zwykłego czasu rzeczywistego

Systemy czasu rzeczywistego dzielą się na: systemy ścisłego czasu rzeczywistego oraz zwykłego czasu rzeczywistego. *System ścisłego czasu rzeczywistego* (ang. *hard real-time system*) wymaga dokładnego przestrzegania granicznych parametrów operacyjnych. Przekroczenie tych parametrów prowadzi do błędu i jest bardzo poważnym problemem. Z drugiej strony, dla *systemu zwykłego czasu rzeczywistego* (ang. *soft real-time system*) przekroczenie granicznych parametrów operacyjnych nie jest błędem krytycznym.

Aplikacje działające w systemach ścisłego czasu rzeczywistego są łatwe do wykrycia: należą do nich systemy ABS, wojskowe systemy obronne, urządzenia medyczne oraz generalnie przetwarzanie sygnałów. Aplikacje zwykłego czasu rzeczywistego nie zawsze można łatwo zidentyfikować. Jednym z oczywistych elementów tej grupy są aplikacje przetwarzające dane wideo: użytkownik zauważa chwilowe pogorszenie jakości, gdy graniczne parametry operacyjne są przekroczone, lecz tych kilka pominiętych ramek obrazu może być w tym momencie tolerowane.

Wiele innych aplikacji posiada ograniczenia czasowe, które (jeśli nie są przestrzegane) pogarszają wrażenia użytkownika. Przykładami są tu aplikacje multimedialne, gry oraz programy sieciowe. A co z edytorami tekstu? Jeśli program nie może szybko reagować na naciśnięcia klawiszy, odpowiedź użytkownika na to zjawisko jest negatywna. Czy jest to więc aplikacja zwykłego czasu rzeczywistego? Oczywiście, gdy projektanci tworzyli tę aplikację, zdawali sobie sprawę, iż reakcja na naciskanie klawiszy musi być możliwie szybka i determinowana czasowo. Ale czy jest to już granicznym parametrem operacyjnym? Granica pomiędzy systemami zwykłego czasu rzeczywistego a systemami standardowymi nie jest wyraźna.

W przeciwieństwie do powszechnego przekonania system czasu rzeczywistego nie musi być szybki. Jeśli uwzględni się porównywalny sprzęt, można stwierdzić, że system czasu rzeczywistego jest prawdopodobnie wolniejszy niż zwykły system, co najmniej z powodu zwiększonego nakładu pracy na wsparcie procesów czasu rzeczywistego. Ponadto, podział pomiędzy systemami ścisłego i zwykłego czasu rzeczywistego nie zależy od wielkości granicznych parametrów operacyjnych. Reaktor atomowy przegrzeje się, jeśli system zabezpieczający przed przegrzaniem nie opuści prętów kontrolnych w ciągu kilku sekund po wykryciu nadmiernego strumienia neutronów. Jest to system ścisłego czasu rzeczywistego ze stosunkowo długim czasem (jak na komputery) parametru granicznego. Przeciwnie, odtwarzacz wideo może opuścić ramkę obrazu lub przerwać dźwięk, jeśli bufor odtwarzania nie będzie wypełniony w ciągu 100 milisekund. Jest to system zwykłego czasu rzeczywistego z krytycznym parametrem granicznym.

Opóźnienie, rozsynchronizowanie oraz parametry graniczne

Termin *opóźnienie* (ang. *latency*) odnosi się do okresu między wystąpieniem zdarzenia a momentem reakcji na to zdarzenie. Jeśli opóźnienie jest mniejsze lub równe wartości parametru granicznego, system działa poprawnie. W wielu systemach ścisłego czasu rzeczywistego operacyjny parametr graniczny oraz opóźnienie są sobie równe — system obsługuje zdarzenia w ściśle ustalonych przedziałach czasowych, w określonych momentach. W systemach zwykłego czasu rzeczywistego żądany czas odpowiedzi jest mniej dokładny i opóźnienie wykazuje pewien rodzaj rozbieżności — należy dążyć do tego, aby odpowiedź mieściła się w zakresie założonego parametru granicznego.

Pomiar opóźnienia jest często trudny, gdyż jego obliczenie wymaga danych na temat czasu, kiedy nastąpiło zdarzenie. Zdolność do oznaczenia dokładnego momentu wystąpienia zdarzenia często jednak pogarsza zdolność poprawnej reakcji na niego. Dlatego też wiele metod obliczania opóźnienia nie działa w ten sposób: zamiast tego mierzy się odchylenie synchronizacji pomiędzy odpowiedziami na zdarzenia. Odchylenie synchronizacji pomiędzy kolejnymi zdarzeniami nazywane jest *rozsynchronizowaniem* (ang. *jitter*), a nie opóźnieniem.

Na przykład, można rozważyć zdarzenie występujące co 10 milisekund. Aby uzyskać charakterystykę sprawności w takim systemie, należałoby mierzyć czasy wystąpień odpowiedzi na te zdarzenia, by upewnić się, że występują faktycznie co 10 milisekund. Odchylenie od tej wartości nie jest jednak opóźnieniem, lecz rozsynchronizowaniem. To, co jest mierzone, jest rozbieżnością kolejnych odpowiedzi. Bez wiedzy o tym, kiedy wystąpiło zdarzenie, nie można znać rzeczywistej różnicy czasu pomiędzy zdarzeniem a odpowiedzią na nie. Nawet wiedząc, że zdarzenie występuje co 10 milisekund, nie ma się pewności, kiedy nastąpił *pierwszy* przypadek tego zdarzenia. Być może jest to zaskakujące, ale wiele metod pomiaru opóźnienia działa błędnie i zwraca wartość rozsynchronizowania, a nie opóźnienia. Oczywiście rozsynchronizowanie to też użyteczny parametr i takie pomiary na pewno są przydatne. Mimo to należy nazywać rzeczy po imieniu!

Systemy ścisłego czasu rzeczywistego często wykazują bardzo małe rozsynchronizowanie, ponieważ odpowiadają na zdarzenie po, a *nie w zakresie* określonego przedziału czasowego. Takie systemy dążą do wartości rozsynchronizowania równej zeru, a opóźnienie równe jest czasowi granicznego opóźnienia operacyjnego. Jeśli to opóźnienie przekracza graniczną wartość opóźnienia operacyjnego, system przestaje działać.

Systemy zwykłego czasu rzeczywistego są bardziej wrażliwe na opóźnienia. W nich czas odpowiedzi znajduje się dokładnie w zakresie dopuszczalnej wartości opóźnienia operacyjnego — nierzadko jest krótszy, czasami dłuższy. W pomiarach sprawności systemu wartość rozsynchronizowania jest często doskonałym zamiennikiem dla parametru opóźnienia.

Wspieranie czasu rzeczywistego przez system Linux

Linux umożliwia aplikacjom korzystanie ze wsparcia dla systemu zwykłego czasu rzeczywistego poprzez rodzinę funkcji systemowych zdefiniowanych przez *IEEE Std 1003.1b-1993* (nazwa często jest skracana do *POSIX 1993* lub *POSIX.1b*).

Mówiąc językiem technicznym, standard POSIX nie wymusza tego, by udostępniany system czasu rzeczywistego był zwykły lub ścisły. W rzeczywistości to, co naprawdę robi standard

POSIX, jest opisem kilku strategii szeregowania, które biorą pod uwagę priorytety. Od projektantów systemu operacyjnego zależy to, jakie metody ograniczeń czasowych zostaną wymuszone przez system dla tych strategii.

Przez lata jądro Linuksa ulepszało swoje wsparcie dla systemu czasu rzeczywistego, dostarczając coraz mniejsze opóźnienia oraz bardziej spójną wartość rozsynchronizowania, przy jednocześnie niezmniejszonej wydajności systemu. Zawdzięczamy te zmiany w większości temu, iż poprawienie opóźnienia wpłynęło pozytywnie na działanie wielu innych grup aplikacji, na przykład procesów powiązanych ze środowiskiem graficznym czy też wejściem i wyjściem, a nie wyłącznie samych aplikacji czasu rzeczywistego. Ulepszenia związane są także z sukcesem Linuksa na polu systemów *wbudowanych* (ang. *embedded*) oraz systemów czasu rzeczywistego.

Niestety wiele z tych modyfikacji, które zostały wprowadzone w jądrze Linuksa dla celów systemów wbudowanych oraz czasu rzeczywistego, istnieje tylko w niestandardowych rozwiązaniach, poza podstawową i oficjalną wersją jądra. Niektóre z tych ulepszeń zapewniają dalsze ograniczenia opóźnienia, a nawet zachowanie charakterystyczne dla systemów ścisłego czasu rzeczywistego. W kolejnych podrozdziałach omówione zostaną tylko oficjalne interfejsy oraz zachowanie standardowego jądra. Na szczęście większość zmian dotyczących wspierania czasu rzeczywistego opiera się w dalszym ciągu na interfejsach POSIX. Zatem dalsze dyskusje odnoszą się również do systemów zmodyfikowanych.

Linuksowe strategie szeregowania i ustalania priorytetów

Zachowanie zarządcy Linuksa w odniesieniu do procesu zależy od *strategii szeregowania* (ang. *scheduling policy*) dla tego procesu, zwanej również *klasą szeregowania* (ang. *scheduling class*). Jako uzupełnienie zwykłej, domyślnej strategii Linux dostarcza dwie strategie szeregowania dla czasu rzeczywistego. Makra preprocesora z pliku nagłówka `<sched.h>` reprezentują każdą z tych strategii i zwane są `SCHED_FIFO`, `SCHED_RR` oraz `SCHED_OTHER`.

Każdy proces posiada *priorytet statyczny* (ang. *static priority*), niezależny od wartości poziomu uprzejmości. Dla zwykłych aplikacji priorytet ten jest zawsze równy zeru. Dla procesów czasu rzeczywistego wynosi on od 1 do 99 włącznie. Linuksowy zarządca zawsze wybiera proces o najwyższym priorytecie, aby go uruchomić (tzn. ten, który posiada priorytet statyczny posiadający największą wartość liczbową). Jeśli proces działa z wartością priorytetu statycznego równą 50, a kolejny proces o priorytecie 51 staje się uruchamialny, zarządca natychmiast wykonuje przeszeregowanie poprzez przełączenie się na tenże uruchamialny proces. I odwrotnie, jeśli w systemie działa już proces o priorytecie 50, a proces o priorytecie 49 staje się uruchamialny, zarządca nie uruchomi go, dopóki proces o priorytecie 50 nie zablokuje się, czyli przestanie działać. Ponieważ normalne procesy mają priorytet 0, każdy proces czasu rzeczywistego, który jest uruchamialny, zawsze wywłaszczy zwykły proces i zacznie działać.

Strategia FIFO (first in, first out)

Klasa „pierwszy na wejściu, pierwszy na wyjściu” (ang. *first in, first out class*), inaczej zwana też *klasą FIFO* (ang. *FIFO class*), jest bardzo prostą strategią czasu rzeczywistego bez przedziałów czasowych. Proces z klasy FIFO będzie działać tak długo, jak długo nie będzie istnieć żaden inny proces o większym priorytecie. Klasa FIFO jest opisywana przez makro `SCHED_FIFO`.

Ponieważ ta strategia nie używa przedziałów czasowych, zasady działań są raczej proste:

- Proces uruchamialny klasy FIFO będzie zawsze działać, jeśli jest procesem o najwyższym priorytecie w systemie. Gdy tylko proces klasy FIFO stanie się uruchamialny, od razu wywłaszczy proces zwykły.
- Proces klasy FIFO będzie kontynuować swoje działanie, dopóki się nie zablokuje lub wywoła `sched_yield()` albo dopóki proces o wyższym priorytecie nie stanie się uruchamialny.
- Gdy proces klasy FIFO zablokuje się, zarządca usunie go z listy procesów uruchamialnych. Gdy znów staje się on uruchamialny, jest umieszczany na końcu listy procesów odpowiadającej jego priorytetowi. Dlatego też nie będzie on uaktywniony, dopóki wszystkie inne procesy o wyższym lub *równym* priorytecie nie przerwą swojego działania.
- Gdy proces klasy FIFO wywoła funkcję `sched_yield()`, zarządca przesunie go na koniec listy procesów odpowiadającej jego priorytetowi. Dlatego też nie będzie on uruchomiony, dopóki wszystkie inne procesy o równym jemu priorytecie nie zakończą swego działania. Jeśli proces wywołujący funkcję `sched_yield()` będzie jedynym procesem posiadającym taki priorytet, wywołanie tej funkcji systemowej nie odniesie żadnego skutku.
- Gdy proces o wyższym priorytecie wywłaszczy proces klasy FIFO, pozostanie on na takiej samej pozycji w liście procesów odpowiadającej jego priorytetowi. Zatem gdy proces o wyższym priorytecie zostanie zawieszony, wywłaszczony proces klasy FIFO będzie mógł w dalszym ciągu kontynuować swoje działanie.
- Gdy proces przystępuje do klasy FIFO lub gdy priorytet statyczny procesu zmienia się, jest on umieszczany na początku listy procesów pasującej do jego priorytetu. W rezultacie proces klasy FIFO otrzymujący nowy priorytet może wywłaszczyć inny działający proces posiadający ten sam priorytet.

Zasadniczo można powiedzieć, iż procesy klasy FIFO działają tak długo, jak chcą, dopóki są procesami posiadającymi najwyższy priorytet w systemie. Ciekawe reguły dotyczą natomiast tych procesów klasy FIFO, które posiadają identyczne priorytety.

Strategia cykliczna

Klasa cykliczna (ang. *round-robin class*) jest identyczną klasą jak FIFO, z wyjątkiem tego, że posiada dodatkowe reguły w przypadku zarządzania procesami o takim samym priorytecie. Klasa cykliczna jest opisywana przez makro `SCHED_RR`.

Zarządca przypisuje każdemu procesowi klasy cyklicznej przedział czasowy. Gdy proces klasy cyklicznej wykorzysta swój przedział czasu, zarządca przesuwa go na koniec listy procesów odpowiadającej jego priorytetowi. W ten sposób procesy klasy cyklicznej, posiadające dany priorytet, są regularnie szeregowane w pętli. Jeśli istnieje tylko jeden proces o danym priorytecie, wówczas klasa cykliczna działa identycznie jak klasa FIFO. W takim przypadku, gdy przedział czasowy ulegnie zużyciu, proces po prostu wznawia swoje działanie.

Można uważać proces klasy cyklicznej za identyczny do klasy FIFO, z wyjątkiem tego, iż dodatkowo wstrzymuje on swoje działanie, gdy zużyje przedział czasowy należący do niego, kiedy jest przenoszony na koniec listy procesów odpowiadającej jego priorytetowi.

Decyzja, czy należy użyć klasy `SCHED_FIFO` lub `SCHED_RR` jest kwestią dotyczącą zachowania się samych priorytetów. Przedziały czasowe klasy cyklicznej mają znaczenie tylko pomiędzy procesami o takim samym priorytecie. Procesy klasy FIFO będą działać niezagrożone; procesy

klasy cyklicznej będą wywłaszczać się pomiędzy sobą w przypadku wystąpień takich samych priorytetów. W żadnym przypadku proces o mniejszym priorytecie nie będzie mógł zostać uruchomiony, gdy działa proces o wyższym priorytecie.

Strategia zwykła

Makro `SCHED_OTHER` opisuje standardową strategię szeregowania, będącą domyślną strategią dla klasy niespełniającej wymagań czasu rzeczywistego. Wszystkie procesy *klasy zwykłej* (ang. *normal class*) posiadają priorytet statyczny równy zeru. Dlatego też każdy uruchamialny proces należący do klasy FIFO lub klasy cyklicznej będzie wywłaszczać proces działający w klasie normalnej.

Zarządca używa poziomów uprzejmości, aby ustalać priorytety procesów wewnątrz klasy zwykłej. Poziom uprzejmości nie oddziałuje na priorytet statyczny, który dla tej klasy pozostaje zawsze równy zeru.

Strategia szeregowania wsadowego

Makro `SCHED_BATCH` opisuje *strategię szeregowania wsadowego* (ang. *batch scheduling policy*), inaczej zwaną też *strategią jałowego szeregowania* (ang. *idle scheduling policy*). Jej zachowanie jest jakby przeciwieństwem strategii czasu rzeczywistego: procesy należące do tej klasy mogą działać tylko wtedy, gdy nie istnieją żadne inne uruchamialne procesy w systemie, nawet jeśli te procesy wykorzystały już swoje przedziały czasowe. Różni się to od zachowania w przypadku procesów z największymi wartościami poziomów uprzejmości (tzn. procesów o najniższych priorytetach) — te procesy w końcu zaczną działać, gdy procesy o wyższych priorytetach wykorzystają swoje przedziały czasowe.

Ustalanie strategii szeregowania dla systemu Linux

Procesy mogą zmieniać strategię szeregowania Linuksa poprzez wywołanie funkcji systemowych `sched_getscheduler()` oraz `sched_setscheduler()`:

```
#include <sched.h>

struct sched_param
{
   /* ... */
   int sched_priority;
   /* ... */
};
int sched_getscheduler (pid_t pid);
int sched_setscheduler (pid_t pid, int policy, const struct sched_param *sp);
```

Poprawne wywołanie funkcji `sched_getscheduler()` odczytuje strategię szeregowania dla procesu określonego poprzez parametr `pid`. Jeśli `pid` wynosi `0`, funkcja zwraca strategię szeregowania dla procesu ją wywołującego. Liczba całkowita zdefiniowana w `<sched.h>` opisuje strategię szeregowania: strategia FIFO określona jest przez `SCHED_FIFO`, strategia cykliczna przez `SCHED_RR`, a strategia normalna przez `SCHED_OTHER`. W przypadku błędu funkcja zwraca `−1` (co nie jest żadną poprawną wartością strategii szeregowania) i odpowiednio ustawia `errno`.

Użycie tej funkcji jest proste:

```
int policy;
/* pobierz strategię szeregowania */
policy = sched_getscheduler (0);
```

```
switch (policy)
{
   case SCHED_OTHER:
      printf ("Strategia normalna\n");
      break;
   case SCHED_RR:
      printf ("Strategia cykliczna\n");
      break;
   case SCHED_FIFO:
      printf ("Strategia FIFO\n");
      break;
   case -1:
      perror ("sched_getscheduler");
      break;
   default:
      fprintf (stderr, "Strategia nieznana!\n");
}
```

Wywołanie funkcji `sched_setscheduler()` ustawia strategię szeregowania dla procesu wskazanego przez `pid` na strategię o wartości przekazanej w parametrze `policy`. Dodatkowe parametry związane ze strategią ustawiane są poprzez parametr `sp`. W przypadku poprawnego wywołania funkcja zwraca `0`, natomiast w przypadku błędu zwraca `−1` oraz odpowiednio ustawia `errno`.

Poprawność pól wewnątrz struktury `sched_param` zależy od strategii szeregowania zapewnianych przez system operacyjny. Strategie `SCHED_RR` oraz `SCHED_FIFO` wymagają jednego pola — `sched_priority`, które reprezentuje priorytet statyczny. `SCHED_OTHER` nie używa żadnego pola, natomiast strategie, które mogą zostać użyte w przyszłości, będą być może wymagać nowych pól. Dlatego też przenośne i poprawne programy nie mogą czynić żadnych założeń dotyczących formatu tej struktury.

Ustawianie strategii oraz parametrów szeregowania dla procesu jest proste:

```
struct sched_param sp;
sp.sched_priority = 1;
int ret;
ret = sched_setscheduler (0, SCHED_RR, &sp);
if (ret == -1)
{
   perror ("sched_setscheduler");
   return 1;
}
```

Ten fragment kodu ustawia strategię szeregowania dla procesu wywołującego na strategię cykliczną oraz ustala priorytet statyczny równy 1. Zakłada się w tym momencie, iż wartość 1 jest poprawną wartością priorytetu — z technicznego punktu widzenia nie zawsze tak musi być. W następnym podrozdziale zostanie omówione, jak należy określać prawidłowy zakres priorytetów dla danej strategii.

Ustawianie strategii innej niż `SCHED_OTHER` wymaga posiadania uprawnienia `CAP_SYS_NICE`. Dlatego też najczęściej administrator ma prawo do uruchamiania procesów czasu rzeczywistego. Od wersji jądra 2.6.12 parametr ograniczeń zasobów `RLIMIT_RTPRIO` pozwala także innym użytkownikom na ustawianie strategii czasu rzeczywistego aż do pewnej określonej granicy priorytetów.

Kody błędów

W przypadku błędu możliwe są cztery wartości `errno`:

EFAULT
: Zmienna wskaźnikowa `sp` wskazuje na błędny lub niedostępny obszar pamięci.

EINVAL
: Strategia szeregowania określona przez parametr `policy` jest nieprawidłowa albo też wartość przekazywana przez `sp` nie ma sensu dla danej strategii (tylko dla `sched_setsche` ↪`duler()`).

EPERM
: Proces wywołujący nie posiada odpowiednich uprawnień.

ESRCH
: Wartość przekazana w `pid` nie odzwierciedla żadnego istniejącego procesu.

Ustawianie parametrów szeregowania

Funkcje systemowe `sched_getparam()` oraz `sched_setparam()`, zdefiniowane przez POSIX, odczytują oraz ustawiają parametry związane z ustaloną już strategią szeregowania:

```
#include <sched.h>
struct sched_param
{
   /* ... */
   int sched_priority;
   /* ... */
};
int sched_getparam (pid_t pid, struct sched_param *sp);
int sched_setparam (pid_t pid, const struct sched_param *sp);
```

Funkcja `sched_getscheduler()` zwraca wyłącznie strategię szeregowania, natomiast nie zwraca związanych z nią parametrów. Wywołanie `sched_getparam()` dla procesu określonego w parametrze `pid` zwraca parametry szeregowania w zmiennej `sp`:

```
struct sched_param sp;
int ret;
ret = sched_getparam (0, &sp);
if (ret == -1)
{
   perror ("sched_getparam");
   return 1;
}
printf ("Nasz priorytet wynosi %d\n", sp.sched_priority);
```

Jeśli `pid` wynosi `0`, funkcja podaje parametry dla procesu wywołującego. W przypadku sukcesu funkcja zwraca `0`, w przypadku niepowodzenia zwraca `-1` oraz odpowiednio ustawia `errno`.

Ponieważ `sched_setscheduler()` również ustawia parametry szeregowania, `sched_setparam()` jest użyteczna tylko w celu późniejszych modyfikacji tych parametrów:

```
struct sched_param sp;
int ret;
sp.sched_priority = 1;
ret = sched_setparam (0, &sp);
if (ret == -1)
{
   perror ("sched_setparam");
   return 1;
}
```

W przypadku sukcesu parametry szeregowania dla procesu `pid` są ustawiane poprzez zmienną `sp`, a funkcja zwraca `0`. W przypadku niepowodzenia funkcja zwraca `-1` oraz odpowiednio ustawia `errno`.

Po uruchomieniu tych dwóch fragmentów kodu otrzyma się następujący wynik:

```
Nasz priorytet wynosi 1
```

Przykład ten także jest oparty na założeniu, że 1 określa prawidłową wartość priorytetu. Tak przeważnie jest, lecz podczas tworzenia programów przenośnych należy to wcześniej sprawdzić. Za chwilę zostanie pokazane, jak ustalać zakres poprawnych priorytetów.

Kody błędów

W przypadku błędu możliwe są cztery wartości `errno`:

`EFAULT`
: Zmienna wskaźnikowa `sp` wskazuje na błędny lub niedostępny obszar pamięci.

`EINVAL`
: Wartość przekazana przez `sp` nie ma sensu dla danej strategii.

`EPERM`
: Proces wywołujący nie posiada niezbędnych uprawnień.

`ESRCH`
: Wartość przekazana w `pid` nie odpowiada żadnemu istniejącemu procesowi.

Określanie zakresu poprawnych priorytetów

Poprzednie przykłady kodów przekazywały sztywno ustalone wartości priorytetów do odpowiednich funkcji systemowych. POSIX nie gwarantuje, że dane wartości priorytetów szeregowania istnieją w określonym systemie, z wyjątkiem tego, że muszą istnieć przynajmniej 32 różne priorytety pomiędzy najniższą a najwyższą wartością. Jak wspomniano już wcześniej w podrozdziale Linuksowe strategie szeregowania i ustalania priorytetów, Linux przypisuje zakres wartości od 1 do 99 włącznie dla dwóch strategii czasu rzeczywistego. Poprawny i przenośny program zwykle definiuje swój własny obszar wartości priorytetów i mapuje je w odpowiedni zakres, właściwy dla systemu operacyjnego. Na przykład, jeśli zaistnieje potrzeba uruchomienia procesów o czterech różnych poziomach priorytetów czasu rzeczywistego, można dynamicznie określić zakres priorytetów i wybrać cztery konkretne wartości.

Linux udostępnia dwie funkcje systemowe dla odczytu zakresu poprawnych wartości priorytetów. Jedna z nich zwraca wartość minimalną, a druga maksymalną:

```
#include <sched.h>
int sched_get_priority_min (int policy);
int sched_get_priority_max (int policy);
```

W przypadku sukcesu wywołanie funkcji `sched_get_priority_min()` zwraca wartość minimalną, natomiast wywołanie `sched_get_priority_max()` zwraca maksymalny poprawny priorytet powiązany ze strategią szeregowania, przekazaną w parametrze `policy`. W przypadku niepowodzenia obie funkcje zwracają `-1`. Jedyny możliwy błąd, jaki może wystąpić, dotyczy błędnego parametru `policy` — wówczas `errno` ustawiane jest na `EINVAL`.

Użycie funkcji jest proste:

```
int min, max;
min = sched_get_priority_min (SCHED_RR);
if (min == -1)
{
   perror ("sched_get_priority_min");
   return 1;
}
max = sched_get_priority_max (SCHED_RR);
if (max == -1)
{
   perror ("sched_get_priority_max");
   return 1;
}
printf ("Zakres wartości priorytetów dla strategii SCHED_RR wynosi: %d - %d\n", min,
max);
```

W standardowym systemie Linux powyższy fragment kodu spowoduje wyświetlenie następującego wyniku:

```
Zakres wartości priorytetów dla strategii SCHED_RR wynosi: 1 - 99
```

Jak już wcześniej wspominano, większe wartości liczbowe oznaczają wyższe priorytety. Aby przypisać procesowi najwyższy priorytet dla jego strategii szeregowania, należy posłużyć się następującym kodem:

```
/*
* set_highest_priority — dla procesu pid ustawia priorytet
* szeregowania na największą wartość, dopuszczaną przez jego
* aktualną strategię szeregowania.
* Jeśli wartość pid wynosi zero, następuje ustawienie priorytetu
* dla aktualnego procesu.
*
* W przypadku sukcesu funkcja zwraca zero.
*/
int set_highest_priority (pid_t pid)
{
   struct sched_param sp;
   int policy, max, ret;

   policy = sched_getscheduler (pid);
   if (policy == -1)
      return -1;

   max = sched_get_priority_max (policy);
   if (max == -1)
      return -1;

   memset (&sp, 0, sizeof (struct sched_param));
   sp.sched_priority = max;
   ret = sched_setparam (pid, &sp);

   return ret;
}
```

Programy zwykle zwracają minimalną lub maksymalną wartość dla systemu, a następnie modyfikują te wielkości o 1 (np. `max - 1`, `max - 2` itd.), aby odpowiednio przypisać priorytety procesom.

sched_rr_get_interval()

Jak już zostało wcześniej powiedziane, procesy klasy SCHED_RR zachowują się tak samo jak procesy klasy SCHED_FIFO, z wyjątkiem tego, że zarządca przydziela tym procesom przedziały czasowe. Gdy proces należący do klasy SCHED_RR wyczerpie swój przedział czasowy, zarządca przesuwa go na koniec listy procesów działających, zawierającej również inne procesy o priorytecie równym jego aktualnemu priorytetowi. W ten oto sposób wszystkie procesy klasy SCHED_RR o tym samym priorytecie są wykonywane cyklicznie. Te o wyższym priorytecie (oraz procesy klasy SCHED_FIFO o takim samym lub wyższym priorytecie) będą zawsze wywłaszczać działający proces klasy SCHED_RR, bez względu na to, czy pozostał mu jakiś przedział czasowy do wykorzystania.

POSIX definiuje interfejs w celu odczytania wielkości przedziału czasowego dla danego procesu:

```
#include <sched.h>
struct timespec
{
   time_t tv_sec;    /* liczba sekund */
   long   tv_nsec;   /* liczba nanosekund */
};
int sched_rr_get_interval (pid_t pid, struct timespec *tp);
```

Poprawne wywołanie funkcji systemowej o skomplikowanej nazwie sched_rr_get_interval() zapisuje w strukturze timespec wskazywanej przez parametr tp czas trwania przedziału czasowego przydzielonego dla procesu pid oraz zwraca zero. W przypadku błędu funkcja zwraca −1 oraz odpowiednio ustawia errno.

Zgodnie ze standardem POSIX funkcja ta jest wymagana wyłącznie dla pracy z procesami klasy SCHED_RR. Jednak w systemie Linux może ona udostępniać długość przedziału czasowego dowolnego procesu. Programy przenośne powinny zakładać, że funkcja ta działa tylko z procesami cyklicznymi; programy dedykowane dla Linuksa mogą nadużywać tej funkcji. Oto przykład:

```
struct timespec tp;
int ret;
/* pobierz wielkość przedziału czasowego dla aktualnego zadania */
ret = sched_rr_get_interval (0, &tp);
if (ret == -1)
{
   perror ("sched_rr_get_interval");
   return 1;
}
/* zamień liczbę sekund i nanosekund na milisekundy */
printf ("Nasz kwant czasowy wynosi %.2lf milisekund\n", (tp.tv_sec * 1000.0f) +
  (tp.tv_nsec / 1000000.0f));
```

Jeśli proces działa w klasie FIFO, wartości zmiennych tv_sec oraz tv_nsec wynoszą 0, co symbolizuje w tym przypadku nieskończoność.

Kody błędu

W przypadku błędu wartości errno są następujące:

EFAULT
: Zmienna wskaźnikowa tp wskazuje na błędny lub niedostępny obszar pamięci.

EINVAL
: Wartość przekazana przez pid jest błędna (np. ujemna).

ESRCH
: Wartość przekazana w `pid` jest poprawna, lecz nie odnosi się do żadnego istniejącego procesu.

Środki ostrożności przy pracy z procesami czasu rzeczywistego

Z powodu samej natury procesów czasu rzeczywistego projektanci powinni być ostrożni podczas tworzenia i uruchamiania takich programów. Gdy program czasu rzeczywistego zaczyna się dziwnie zachowywać, cały system może się zawiesić. Każda pętla związana z procesorem w programie czasu rzeczywistego, to znaczy dowolny kawałek kodu, który nie może się zatrzymać, będzie kontynuować swoje działanie w nieskończoność, tak długo, dopóki procesy czasu rzeczywistego o wyższym priorytecie nie staną się uruchamialne.

Dlatego też projektowanie programów czasu rzeczywistego wymaga ostrożności oraz szczególnej uwagi. Takie programy rządzą w sposób absolutny i mogą w prosty sposób spowodować zawieszenie całego systemu. Oto kilka wskazówek i ostrzeżeń:

- Należy pamiętać, że każda pętla programowa wykonywana przez proces związany z procesorem będzie działać bez jakiegokolwiek przerwania pracy, dopóki się naturalnie nie zakończy, jeśli w systemie nie pojawi się żaden proces o wyższym priorytecie. Jeśli pętla jest nieskończona, system zawiesi się.
- Ponieważ procesy czasu rzeczywistego działają kosztem całego systemu, należy zwrócić szczególną uwagę na sposób ich zaprojektowania. Trzeba zadbać, aby nie dopuścić do trwałego zablokowania systemu z powodu poświęcenia mu zbyt małej ilości czasu procesora.
- Należy być bardzo ostrożnym w przypadku oczekiwania w pętli (ang. *busy-wait*). Jeśli proces czasu rzeczywistego oczekuje w pętli na zasób zajmowany przez proces o niższym priorytecie, będzie niestety czekać w nieskończoność.
- Podczas projektowania procesu czasu rzeczywistego zalecane jest używanie terminala, który posiada wyższy priorytet niż proces projektowany. Dzięki temu w przypadku problemu terminal pozostanie w dalszym ciągu aktywny i umożliwi usunięcie procesu, który wymknął się spod kontroli (dopóki terminal pozostanie w stanie jałowym, oczekując na dane z klawiatury, nie będzie przeszkadzać innym procesom).
- Program użytkowy *chrt*, jeden z elementów pakietu narzędziowego *util-linux*, upraszcza odczytywanie i ustawianie atrybutów czasu rzeczywistego dla innych procesów. Narzędzie to ułatwia uruchamianie dowolnych programów w trybie szeregowania czasu rzeczywistego, na przykład wyżej wspomnianego terminala, jak również pozwala na zmianę priorytetów czasu rzeczywistego dla istniejących aplikacji.

Determinizm

Procesy czasu rzeczywistego są pełne determinizmu. W obliczeniach czasu rzeczywistego czynność jest *deterministyczna* (ang. *deterministic*), jeśli przy takich samych danych wejściowych generuje zawsze ten sam wynik w tym samym przedziale czasu. Nowoczesne komputery są bardzo dobrym przykładem systemów niedeterministycznych: wielopoziomowe pamięci podręczne (poddające się trafieniom i chybieniom bez żadnej przewidywalności), wieloprocesorowość, stronicowanie (ang. *paging*), wymiana danych (ang. *swapping*) oraz wielozadaniowość niszczą każde oszacowanie zmierzające do tego, aby ustalić, jak długo dana czynność będzie

się wykonywać. Oczywiście osiągnięto już punkt, w którym praktycznie każda czynność (za wyjątkiem dostępu do twardego dysku) jest „niesamowicie szybka", ale jednocześnie nowoczesne systemy utrudniły dokładne sprecyzowanie, ile czasu zajmie jej wykonanie.

Aplikacje czasu rzeczywistego często próbują ogólnie ograniczyć nieprzewidywalność, a w szczególności najbardziej niekorzystne opóźnienia. Kolejne podrozdziały omówią dwie metody, które używane są w tym celu.

Wcześniejsze zapisywanie danych oraz blokowanie pamięci

Rozważmy następującą sytuację: zostaje wygenerowane przerwanie sprzętowe, pochodzące od niestandardowego monitora śledzącego międzykontynentalne pociski balistyczne, co powoduje, że sterownik urządzenia chce szybko skopiować dane z układu sprzętowego do jądra systemu. Sterownik zauważa jednak, że proces jest uśpiony, blokując się na elemencie sprzętowym i czekając na dane. Powiadamia jądro, aby obudziło ten proces. Jądro, zauważając, że działa on w strategii czasu rzeczywistego i posiada wysoki priorytet, natychmiast wywłaszcza aktualnie działający proces, decydując o natychmiastowym zaszeregowaniu procesu czasu rzeczywistego. Zarządca uruchamia tenże proces i przełącza kontekst do jego przestrzeni adresowej. Proces zaczyna działać. Cała akcja trwa 0,3 milisekundy, przy zakładanej wartości najgorszego przypadku opóźnienia równej 1 milisekundzie.

Proces, będąc obecnie na poziomie użytkownika, zauważa nadlatujący międzykontynentalny pocisk balistyczny i zaczyna przetwarzać jego trajektorię. Gdy balistyka zostanie wyliczona, proces czasu rzeczywistego uruchamia wystrzelenie pocisku służącego do niszczenia głowic rakiet o dalekim zasięgu. Minęło kolejne 0,1 milisekundy — wystarczające, aby uruchomić odpowiedź systemu obronnego i ocalić życie ludzi. Jednak w tym momencie kod zarządzający wystrzeleniem pocisku zostaje przerzucony na dysk! Następuje błąd dostępu do strony, procesor z powrotem przełącza się w tryb jądra, jądro rozpoczyna operacje dyskowe wejścia i wyjścia w celu przeniesienia zapisanych danych z powrotem do pamięci. Zarządca przełącza proces w tryb uśpienia na czas obsługi błędu dostępu. Mijają kolejne sekundy. Jest już za późno…

Stronicowanie i wymiana danych wprowadzają oczywiście całkiem niedeterministyczne zachowanie, które może spowodować zniszczenia w procesie czasu rzeczywistego. Aby uchronić się przed taką katastrofą, aplikacje czasu rzeczywistego często „blokują" lub „wiążą na stałe" wszystkie strony ze swojej przestrzeni adresowej z pamięcią fizyczną, zapisując je wstępnie do niej i chroniąc przed wymieceniem na dysk. Gdy tylko strony zostaną zablokowane w pamięci, jądro już ich nie zapisze na dysk. Dowolny dostęp do tych stron nie spowoduje żadnych błędów. Większość aplikacji czasu rzeczywistego blokuje niektóre lub nawet wszystkie swoje strony w fizycznej pamięci operacyjnej.

Linux dostarcza funkcji systemowych w celu wcześniejszego zapisywania oraz blokowania danych. W rozdziale 4. omówione zostały interfejsy używane dla zapisywania danych do pamięci fizycznej. W rozdziale 8. przedstawione będą funkcje systemowe przeznaczone dla blokowania danych w fizycznej pamięci operacyjnej.

Wiązanie do procesora a procesy czasu rzeczywistego

Drugim problemem aplikacji czasu rzeczywistego jest wielozadaniowość. Choć jądro Linuksa działa w trybie wywłaszczenia, jego zarządca nie zawsze potrafi na żądanie przeszeregować jeden proces, by udostępnić czas drugiemu. Czasami aktualnie działający proces wykonuje

się wewnątrz regionu krytycznego w jądrze i zarządca nie potrafi wywłaszczyć go, dopóki nie opuści on tego regionu. Jeśli jest to proces, który oczekuje na uruchomienie w czasie rzeczywistym, to opóźnienie może być nie do zaakceptowania i istnieje niebezpieczeństwo, że szybko przekroczy graniczny parametr operacyjny.

Zatem wielozadaniowość wprowadza niedeterminizm podobny w naturze do nieprzewidywalności spotykanej w stronicowaniu. Rozwiązanie, które bierze pod uwagę wielozadaniowość, jest jedno: należy ją po prostu wyeliminować. Niestety jest prawdopodobne, że nie da się łatwo usunąć wszystkich innych procesów. Gdyby było to możliwe w danym środowisku, prawdopodobnie nie byłoby potrzeby użycia systemu Linux — wystarczyłby dowolny prosty system operacyjny. Jeśli jednak system posiada wiele procesorów, można przeznaczyć jeden lub więcej z tych procesorów dla procesu (lub procesów) czasu rzeczywistego. W praktyce można chronić procesy czasu rzeczywistego przed wielozadaniowością.

Wcześniej w tym rozdziale omówiono funkcje systemowe, używane w celu kontroli wiązania procesora dla danego procesu. Możliwą optymalizacją dla aplikacji czasu rzeczywistego jest zarezerwowanie po jednym procesorze dla każdego z procesów czasu rzeczywistego oraz zezwolenie wszystkim innym procesom na dzielenie swojego czasu na pozostałym procesorze.

Najprostszą metodą, aby to osiągnąć, jest takie zmodyfikowanie jednej z wersji głównego programu inicjalizującego („ojca procesów") *init* o nazwie *SysVinit*[3], aby wykonywał coś podobnego do kodu zamieszczonego poniżej, zanim rozpocznie proces startowania systemu:

```
cpu_set_t set;
int ret;

CPU_ZERO (&set); /* wyzeruj wszystkie procesory */
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
{
   perror ("sched_getaffinity");
   return 1;
}

CPU_CLR (1, &set); /* nie dopuszczaj procesora o numerze 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
{
   perror ("sched_setaffinity");
   return 1;
}
```

Ten fragment kodu pobiera aktualny zestaw dozwolonych procesorów dla programu *init*. Zestaw ten zawiera (jak się tego oczekuje) wszystkie dostępne procesory. W dalszej części kodu następuje usunięcie z tego zbioru procesora o numerze 1 oraz aktualizacja listy dopuszczonych procesorów.

Ponieważ lista dozwolonych procesorów jest dziedziczona przez potomstwo, a program *init* jest patronem wszystkich procesów, wszystkie procesy systemowe będą działać z takim zestawem dozwolonych procesorów, na jaki zezwoli program *init*. Dlatego też w tym przypadku żaden z procesów nie będzie używać procesora o numerze 1.

[3] Źródła *SysVinit* znajdują się pod adresem *ftp://ftp.cistron.nl/pub/people/miquels/sysvinit/*. Program jest licencjonowany zgodnie z Powszechną Licencją Publiczną GNU v2.

Następnie należy tak zmodyfikować proces czasu rzeczywistego, aby działał wyłącznie na procesorze nr 1:

```
cpu_set_t set;
int ret;

CPU_ZERO (&set); /* wyzeruj wszystkie procesory */
CPU_SET (1, &set); /* pozwalaj tylko na procesor o numerze 1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
{
   perror ("sched_setaffinity");
   return 1;
}
```

Wynikiem użycia powyższych fragmentów kodu jest to, iż procesy czasu rzeczywistego działają tylko na procesorze nr 1, a wszystkie inne procesy używają pozostałych procesorów.

Ograniczenia zasobów systemowych

Jądro Linuksa narzuca na procesy pewne *ograniczenia zasobów systemowych* (ang. *resource limits*). Te ograniczenia ustalają bezwzględne górne granice dotyczące wielkości zasobów jądra, których proces może używać, na przykład liczby otwartych plików, stron pamięci, zawieszonych sygnałów itd. Ograniczenia są stosowane bezwarunkowo; jądro nie zezwoli na działanie, które pozwoliłoby procesowi użyć jakiegoś zasobu powyżej dopuszczalnej górnej granicy. Na przykład, jeśli otwarcie nowego pliku pozwoliłoby procesowi posiadać więcej otwartych plików, niż zezwolono w odpowiednim ograniczeniu zasobów systemowych, wywołanie funkcji `open()` nie powiedzie się[4].

Linux udostępnia dwie funkcje systemowe służące kontroli ograniczeń zasobów systemowych. Co prawda POSIX znormalizował oba interfejsy, lecz Linux wspiera dodatkowo jeszcze inne ograniczenia zasobów, uzupełniając te, które istnieją już zgodnie ze standardem. Ograniczenia mogą być odczytane poprzez wywołanie funkcji `getrlimit()`, a ustawione przez funkcję `setrlimit()`:

```
#include <sys/time.h>
#include <sys/resource.h>

struct rlimit
{
   rlim_t rlim_cur; /* ograniczenie miękkie */
   rlim_t rlim_max; /* ograniczenie twarde */
};
int getrlimit (int resource, struct rlimit *rlim);
int setrlimit (int resource, const struct rlimit *rlim);
```

Zasoby reprezentowane są przez stałe całkowitoliczbowe, takie jak na przykład `RLIMIT_CPU`. Struktura `rlimit` opisuje bieżące ograniczenia. Definiuje dwie górne granice: *ograniczenie miękkie* (ang. *soft limit*) oraz *ograniczenie twarde* (ang. *hard limit*). Jądro wymusza miękkie ograniczenie zasobów na procesach, lecz proces może swobodnie zmieniać je na dowolną wartość w zakresie od zera aż do granicy określanej przez ograniczenie twarde. Proces nieposiadający uprawnienia

[4] W tym przypadku wywołana funkcja systemowa ustawi `errno` na wartość `EMFILE`, informując, iż proces osiągnął graniczną dopuszczalną wartość dla liczby otwartych plików. Funkcja systemowa `open()` omówiona jest w rozdziale 2.

CAP_SYS_RESOURCE (czyli niebędący procesem administratora) może jedynie obniżyć swoje ograniczenie twarde. Proces bez przywilejów nie może nigdy zwiększyć swoich ograniczeń twardych, nawet do wyższej wartości, którą poprzednio posiadał — obniżanie ograniczeń twardych jest nieodwracalne. Proces uprzywilejowany może ustawić swoje ograniczenie twarde na dowolną poprawną wartość.

Od konkretnego zasobu zależy, co naprawdę reprezentuje jego ograniczenie. Jeśli zasobem jest na przykład RLIMIT_FSIZE, wówczas ograniczenie opisuje maksymalny rozmiar pliku w bajtach, który proces może stworzyć. W tym przypadku, jeśli rlim_cur wynosi 1024, proces nie może utworzyć ani powiększyć pliku większego niż rozmiar jednego kilobajta.

Wszystkie ograniczenia zasobów systemowych posiadają dwie specjalne wartości: 0 oraz nieskończoność. Pierwsza wartość zupełnie blokuje użycie danego zasobu. Na przykład, jeśli RLIMIT_CORE jest równe zeru, jądro nigdy nie stworzy pliku zrzutu systemowego. I odwrotnie, druga wartość likwiduje wszelkie ograniczenia dla danego zasobu. Jądro rozpoznaje nieskończoność poprzez specjalną wartość RLIM_INFINITY, która wynosi dokładnie −1 (może to spowodować pewne nieporozumienia, gdyż −1 jest również wartością zwracaną w przypadku błędu). Jeśli RLIMIT_CORE jest równy nieskończoności, jądro będzie tworzyć pliki zrzutu systemowego o dowolnym rozmiarze.

Funkcja getrlimit() pobiera aktualne ograniczenia miękkie oraz twarde dla zasobu określanego poprzez parametr resource i umieszcza je w strukturze opisywanej przez rlim. W przypadku poprawnego wywołania funkcja zwraca 0, natomiast w przypadku błędu zwraca −1 oraz odpowiednio ustawia errno.

Funkcja systemowa setrlimit() ustawia odpowiednio ograniczenia miękkie i twarde dla zasobu resource na wartości wskazywane przez strukturę rlim. W przypadku poprawnego wywołania funkcja zwraca 0, a jądro stosownie uaktualnia ograniczenia zasobów systemowych. W przypadku błędu funkcja zwraca −1 oraz ustawia errno na właściwą wartość.

Ograniczenia

Linux aktualnie udostępnia 15 ograniczeń zasobów systemowych:

RLIMIT_AS
: Ogranicza w bajtach maksymalny rozmiar przestrzeni adresowej procesu. Próba zwiększenia rozmiaru przestrzeni adresowej poza to ograniczenie (poprzez funkcje systemowe takie jak mmap() oraz brk()) nie powiedzie się, a funkcje zwrócą wartość błędu ENOMEM. Jeśli stos procesu, który automatycznie rośnie w miarę potrzeby, przekracza to ograniczenie, jądro wysyła temu procesowi sygnał SIGSEGV. Limit ten wynosi zwykle RLIM_INFINITY.

RLIMIT_CORE
: Ustala maksymalny rozmiar w bajtach pliku zrzutu systemowego (core). Jeśli ograniczenie jest niezerowe, pliki zrzutu systemowego większe od tej wielkości obcinane są do maksymalnego dopuszczanego rozmiaru. Jeśli ograniczenie wynosi zero, pliki zrzutu systemowego nie są nigdy tworzone.

RLIMIT_CPU
: Wyznacza maksymalną ilość czasu procesora w sekundach, która może być zużyta przez proces. Jeśli proces działa dłużej niż to ograniczenie, jądro wysyła procesowi sygnał SIGXCPU, który może być jednak przechwycony i obsłużony przez tenże proces. Programy przenośne

powinny zakończyć swoje działanie po otrzymaniu takiego sygnału, ponieważ POSIX nie definiuje żadnej akcji dla jądra po jego wysłaniu. Jednak Linux zezwala procesom, by kontynuowały swoje działanie, a następnie wysyła regularnie co jedną sekundę kolejne sygnały SIGXCPU. Gdy tylko nastąpi osiągnięcie granicy ograniczenia twardego, do procesu zostaje wysłany sygnał SIGKILL, który powoduje zakończenie jego działania.

RLIMIT_DATA
: Zarządza maksymalnym rozmiarem w bajtach segmentu danych i stosu dla danego procesu. Próba zwiększenia rozmiaru segmentu danych poza to ograniczenie przy pomocy funkcji systemowej brk() kończy się niepowodzeniem i zwraca ENOMEM.

RLIMIT_FSIZE
: Określa maksymalny rozmiar w bajtach dla pliku, który może być stworzony przez dany proces. Jeśli proces spróbuje rozszerzyć plik ponad tę granicę, jądro wyśle mu sygnał SIGXFSZ. Sygnał ten domyślnie kończy działanie procesu. Proces może jednak przechwycić go i obsłużyć, co spowoduje, że kolejne próby wywołania funkcji systemowej poszerzającej rozmiar pliku będą kończyć się niepowodzeniem i zwracać błąd EFBIG.

RLIMIT_LOCKS
: Zarządza maksymalną liczbą blokad pliku, które mogą być w posiadaniu przez dany proces (w rozdziale 7. można zapoznać się z dyskusją na temat blokad pliku). Gdy tylko ograniczenie zostaje osiągnięte, dalsze próby otrzymania dodatkowych blokad pliku powinny kończyć się niepowodzeniem i zwracać błąd ENOLCK. Jądro Linuksa w wersji 2.4.25 likwiduje jednak tę cechę. Obecne wersje jądra pozwalają na ustawienie ograniczenia, lecz nie powoduje to żadnych zmian.

RLIMIT_MEMLOCK
: Określa maksymalny rozmiar w bajtach dla pamięci, która może być zablokowana przy pomocy funkcji systemowych mlock(), mlockall() lub shmctl() przez proces nieposiadający uprawnienia CAP_SYS_IPC (czyli tak naprawdę przez proces niebędący administratorem). Jeśli to ograniczenie zostaje przekroczone, wywołania tych funkcji kończą się błędem i zwracają kod EPERM. W praktyce rzeczywiste ograniczenie jest zaokrąglone w dół, do liczby całkowitej będącej wielokrotnością liczby stron. Procesy posiadające uprawnienie CAP_SYS_IPC mogą zablokować dowolną liczbę stron w pamięci i ograniczenie to nie powoduje w ich przypadku żadnych zmian. Zanim pojawiła się wersja jądra 2.6.9, limit ten dotyczył procesów z uprawnieniem CAP_SYS_IPC, a procesy bez przywilejów w ogóle nie mogły blokować żadnych stron. Ograniczenie to nie jest częścią standardu POSIX, wprowadził je system BSD.

RLIMIT_MSGQUEUE
: Określa maksymalną wielkość obszaru w bajtach, który może być przydzielony przez użytkownika dla potrzeb kolejek wiadomości w standardzie POSIX. Jeśli nowo utworzona kolejka wiadomości przekroczy to ograniczenie, funkcja systemowa mq_open() wykona się niepoprawnie i zwróci kod błędu ENOMEM. Ograniczenie to nie jest częścią standardu POSIX; zostało dodane w jądrze wersji 2.6.8 i jest specyficzne dla Linuksa.

RLIMIT_NICE
: Określa maksymalną wartość, do której dany proces może obniżyć swój poziom uprzejmości (czyli podnieść swój priorytet). Jak zostało to już wcześniej omówione w niniejszym rozdziale, procesy zwykle mogą jedynie zwiększać wartość swojego poziomu uprzejmości (zmniejszać swój priorytet). To ograniczenie umożliwia administratorowi ustalenie maksymalnej wartości (czyli dolnej granicy poziomu uprzejmości), do której procesy mogą

poprawnie zwiększać swój priorytet. Ponieważ poziomy uprzejmości mogą być ujemne, jądro interpretuje wartość jako `20 - rlim_cur`. Zatem jeśli ograniczenie ustawione jest na 40, proces może obniżyć wartość swojego poziomu uprzejmości do minimalnej wartości równej –20 (jest to jednocześnie najwyższy możliwy priorytet). To ograniczenie zostało wprowadzone w jądrze w wersji 2.6.12.

`RLIMIT_NOFILE`
: Ustala liczbę będącą wartością o jeden większą od maksymalnej liczby deskryptorów plików, które dany proces może mieć otwarte. Próba ominięcia tego ograniczenia kończy się błędem i odpowiednia funkcja systemowa zwraca kod `EMFILE`. To ograniczenie jest również znane pod nazwą `RLIMIT_OFILE`, która pochodzi z systemu BSD.

`RLIMIT_NPROC`
: Określa maksymalną liczbę procesów, które mogą działać, podczas gdy są uruchomione w danym momencie przez użytkownika w systemie. Próba przekroczenia tego ograniczenia kończy się niepowodzeniem, a funkcja systemowa `fork()` zwraca kod błędu `EAGAIN`. Ograniczenie to nie jest częścią standardu POSIX, wprowadził je system BSD.

`RLIMIT_RSS`
: Określa maksymalną liczbę stron, które proces może umieścić w pamięci (ta wielkość znana też jest pod nazwą *rozmiaru grupy rezydentnej*, ang. *resident set size* — RSS). Tylko wczesne wersje 2.4 jądra systemu egzekwowały to ograniczenie. Obecne wersje jądra zezwalają co prawda na ustawienie tej wartości, ale nie powoduje to żadnych zmian. To ograniczenie nie jest częścią standardu POSIX; wprowadził je system BSD.

`RLIMIT_RTPRIO`
: Określa maksymalny poziom priorytetu czasu rzeczywistego, który może być zażądany przez proces nieposiadający uprawnienia `CAP_SYS_NICE` (czyli w rzeczywistości proces niebędący administratorem). Zwykle procesom bez przywilejów nie wolno żądać żadnej klasy szeregowania czasu rzeczywistego. Ograniczenie to nie jest częścią standardu POSIX; zostało dodane w wersji jądra 2.6.12 i jest specyficzne dla Linuksa.

`RLIMIT_SIGPENDING`
: Ustala maksymalną liczbę sygnałów (standardowych oraz czasu rzeczywistego), które mogą być umieszczone w kolejce dla danego użytkownika. Próba dodania do kolejki następnych sygnałów nie powiedzie się, a funkcje systemowe takie jak `sigqueue()` zwrócą wartość błędu `EAGAIN`. Należy zwrócić uwagę, że bez względu na to ograniczenie zawsze możliwe jest dodanie do kolejki jednego egzemplarza sygnału o typie, który jeszcze się w niej nie znajduje. Dzięki temu zawsze istnieje możliwość wysłania procesowi sygnałów takich jak `SIGKILL` czy `SIGTERM`. Ograniczenie to nie jest częścią standardu POSIX; jest specyficzne dla Linuksa.

`RLIMIT_STACK`
: Ustala maksymalny rozmiar w bajtach stosu dla procesu. Próba ominięcia tego ograniczenia kończy się wysłaniem sygnału `SIGSEGV`.

Ograniczenia domyślne

Ograniczenia domyślne dostępne dla procesu zależą od trzech zmiennych: wstępnego ograniczenia miękkiego, wstępnego ograniczenia twardego oraz administratora systemu. Jądro narzuca wstępne ograniczenia miękkie i twarde, pokazane w tabeli 6.1. Jądro ustawia te ograniczenia dla procesu *init*, a ponieważ potomkowie dziedziczą ograniczenia od swoich rodziców, wszystkie dalsze procesy przejmują ograniczenia miękkie i twarde od procesu *init*.

Tabela 6.1. Domyślne miękkie i twarde ograniczenia zasobów systemowych

Symbol ograniczenia	Ograniczenie miękkie	Ograniczenie twarde
RLIMIT_AS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_CORE	0	RLIM_INFINITY
RLIMIT_CPU	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_DATA	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_FSIZE	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_LOCKS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_MEMLOCK	8 stron	8 stron
RLIMIT_MSGQUEUE	800 kB	800 kB
RLIMIT_NICE	0	0
RLIMIT_NOFILE	1024	1024
RLIMIT_NPROC	0 (oznacza brak ograniczeń)	0 (oznacza brak ograniczeń)
RLIMIT_RSS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_RTPRIO	0	0
RLIMIT_SIGPENDING	0	0
RLIMIT_STACK	8 MB	RLIM_INFINITY

Dwa przypadki mogą zmienić te domyślne wartości ograniczeń:

- Każdy proces może swobodnie zwiększyć ograniczenie miękkie do dowolnej wartości od 0 do wielkości ograniczenia twardego. Potomkowie będą dziedziczyć te uaktualnione ograniczenia podczas rozwidlenia procesu.
- Proces uprzywilejowany może swobodnie ustawiać ograniczenie twarde na dowolną wartość. Potomkowie odziedziczą te uaktualnione ograniczenia podczas rozwidlenia procesu.

Jest raczej mało prawdopodobne, że prawidłowo utworzony proces administratora zmieni jakieś ograniczenia twarde. Dlatego też pierwszy przypadek jest dużo bardziej prawdopodobnym źródłem modyfikacji ograniczeń niż drugi. Istotnie, faktyczne ograniczenia udostępniane procesowi są w większości ustawiane przez powłokę systemową użytkownika, która może być w taki sposób dopasowana przez administratora, aby ustalać różne restrykcje. Na przykład, w powłoce systemowej **Bourne-again shell** (*bash*) administrator uzyskuje dostęp do parametrów ograniczeń dzięki komendzie *ulimit*. Należy zwrócić uwagę, że administrator nie potrzebuje niższych wartości; może on również podnieść ograniczenia miękkie do poziomu ograniczeń twardych, dostarczając użytkownikom rozsądniejszych wielkości domyślnych. Jest to często stosowane w przypadku ograniczenia `RLIMIT_STACK`, które w wielu systemach ustawia się na wartość `RLIM_INFINITY`.

Ustawianie i odczytywanie ograniczeń

Definicje różnych ograniczeń systemowych zostały już objaśnione, dlatego też można przystąpić do ich odczytywania i ustawiania. Odczytywanie wartości ograniczenia systemowego jest całkiem proste:

```
struct rlimit rlim;
int ret;

/* pobierz ograniczenie dla rozmiarów pliku zrzutu systemowego */
```

```
ret = getrlimit (RLIMIT_CORE, &rlim);
if (ret == -1)
{
   perror ("getrlimit");
   return 1;
}
printf ("Ograniczenie RLIMIT_CORE: miękkie=%ld twarde=%ld\n", rlim.rlim_cur,
↪rlim.rlim_max);
```

Skompilowanie tego fragmentu kodu i jego uruchomienie udostępnia następujący wynik:

```
Ograniczenie RLIMIT_CORE: miękkie=0 twarde=-1
```

Ograniczenie miękkie ustawione jest na `0`, natomiast twarde na nieskończoność (`-1` oznacza `RLIM_INFINITY`). Dlatego też możliwe jest ustawienie limitu miękkiego na dowolną wartość. Poniższy przykład ustawia maksymalny rozmiar pliku zrzutu systemowego na 32 MB:

```
struct rlimit rlim;
int ret;
rlim.rlim_cur = 32 * 1024 * 1024; /* 32 MB */
rlim.rlim_max = RLIM_INFINITY; /* zostawiamy tak, jak jest */
ret = setrlimit (RLIMIT_CORE, &rlim);
if (ret == -1)
{
   perror ("setrlimit");
   return 1;
}
```

Kody błędów

W przypadku błędów możliwe są trzy wartości `errno`:

`EFAULT`
: Obszar pamięci wskazywany przez `rlim` jest błędny lub niedostępny.

`EINVAL`
: Wartość określona przez parametr `resource` jest nieprawidłowa albo wartość `rlim.rlim_cur` jest większa od `rlim.rlim_max` (tylko dla `setrlimit()`).

`EPERM`
: Proces wywołujący nie posiadał uprawnień `CAP_SYS_RESOURCE`, a próbował zwiększyć ograniczenie twarde.

ROZDZIAŁ 7.

Zarządzanie plikami i katalogami

W rozdziałach od drugiego do czwartego przedstawiono wiele metod obsługi operacji wejścia i wyjścia. W tym rozdziale ponownie omówione zostaną pliki, tym razem jednak analizie nie będą poddane operacje czytania i zapisywania w nich, lecz raczej ich modyfikowanie oraz zarządzanie nimi i ich metadanymi.

Pliki i ich metadane

Jak napisano w rozdziale 1., każdy plik związany jest z *i-węzłem*, który oznaczany jest unikalną dla danego systemu plików wartością liczbową, zwaną *numerem i-węzła*. I-węzeł jest obiektem fizycznym, umieszczonym na dysku w uniksopodobnym systemie plików, a zarazem jednostką pojęciową, reprezentowaną przez strukturę danych jądra Linuksa. I-węzeł przechowuje *metadane* związane z plikiem, takie jak uprawnienia dostępu, czas ostatniego dostępu, nazwę właściciela, nazwę grupy właścicielskiej, długość, jak również położenie danych.

Numer i-węzła dla pliku można uzyskać poprzez użycie opcji *-i* w poleceniu *ls*:

```
$ ls -i
1689459 Kconfig     1689461 main.c       1680144 process.c    1689464 swsusp.c
1680137 Makefile    1680141 pm.c         1680145 smp.c        1680149 user.c
1680138 console.c   1689462 power.h      1689463 snapshot.c
1689460 disk.c      1680143 poweroff.c   1680147 swap.c
```

W powyższych wynikach wykonania polecenia *ls* można odnaleźć przykładową informację, że plik *disk.c* posiada numer i-węzła równy 1689460. W tym określonym systemie plików nie istnieje żaden inny plik posiadający taki sam numer i-węzła. Nie można jednak tego zagwarantować dla innego systemu plików.

Rodzina funkcji stat

Unix udostępnia rodzinę funkcji, pozwalających na odczytanie metadanych z danego pliku:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
int stat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *path, struct stat *buf);
```

Każda z tych funkcji zwraca informacje o pliku. Funkcja `stat()` zwraca informacje o pliku, dla którego ścieżka podana jest w parametrze `path`, natomiast funkcja `fstat()` zwraca informacje o pliku reprezentowanym przez deskryptor pliku `fd`. Funkcja `lstat()` jest identyczna z funkcją `stat()`, za wyjątkiem tego, że w przypadku dowiązania symbolicznego zwraca informacje o nim, a nie o pliku docelowym.

Każda z tych funkcji przechowuje zwrócone informacje w strukturze `stat`, która dostarczana jest przez użytkownika. Struktura `stat` zdefiniowana jest w pliku nagłówkowym `<bits/stat.h>`, który z kolei dołączony jest do pliku `<sys/stat.h>`:

```
struct stat
{
   dev_t st_dev; /* identyfikator urządzenia zawierającego plik */
   ino_t st_ino; /* numer i-węzła */
   mode_t st_mode; /* uprawnienia */
   nlink_t st_nlink; /* liczba dowiązań twardych */
   uid_t st_uid; /* identyfikator użytkownika będącego właścicielem pliku */
   gid_t st_gid; /* identyfikator grupy właścicielskiej pliku */
   dev_t st_rdev; /* identyfikator urządzenia (w przypadku, gdy plik jest plikiem specjalnym) */
   off_t st_size; /* całkowity rozmiar w bajtach */
   blksize_t st_blksize; /* rozmiar bloku dla plikowych operacji wejścia i wyjścia */
   blkcnt_t st_blocks; /* liczba przydzielonych bloków */
   time_t st_atime; /* czas ostatniego dostępu */
   time_t st_mtime; /* czas ostatniej modyfikacji */
   time_t st_ctime; /* czas ostatniej zmiany statusu */
};
```

Poniżej przedstawiony jest bardziej szczegółowy opis pól:

- Pole `st_dev` oznacza węzeł urządzenia, w którym znajduje się plik (węzły urządzeń zostaną omówione w dalszej części tego rozdziału). Jeśli plik nie jest związany z urządzeniem — na przykład, jeśli znajduje się w zamontowanym systemie plików NFS — wartość ta jest równa zeru.
- Pole `st_ino` udostępnia numer i-węzła dla pliku.
- Pole `st_mode` udostępnia dane przechowujące prawa dostępu do pliku. Definicje uprawnień omówione zostały w rozdziałach pierwszym i drugim.
- Pole `st_nlink` udostępnia liczbę dowiązań twardych wskazujących na dany plik. Każdy plik posiada co najmniej jedno dowiązanie twarde.
- Pole `st_uid` udostępnia identyfikator użytkownika, który jest właścicielem pliku.
- Pole `st_gid` udostępnia identyfikator grupy właścicielskiej, która posiada dany plik.
- Jeśli plik jest węzłem urządzenia, wówczas pole `st_rdev` opisuje urządzenie reprezentowane przez tenże plik.
- Pole `st_size` udostępnia rozmiar pliku w bajtach.
- Pole `st_blksize` definiuje zalecany rozmiar bloku dla wydajnych operacji plikowych wejścia i wyjścia. Wartość ta (lub jej całkowita wielokrotność) jest optymalnym rozmiarem bloku dla operacji wejścia i wyjścia, buforowanych w przestrzeni użytkownika (dokładny opis znajduje się w rozdziale 3.).

- Pole `st_blocks` udostępnia liczbę bloków w systemie plików, które zostały przydzielone dla pliku. Jeśli plik posiada luki (czyli jest plikiem rzadkim), wówczas wartość ta będzie mniejsza od wartości, udostępnionej w polu `st_size`.
- Pole `st_atime` zawiera czas ostatniego dostępu do pliku. Zapisany jest tu czas ostatniej operacji, podczas której plik został udostępniony (przeprowadzonej na przykład przy użyciu funkcji `read()` lub `execle()`).
- Pole `st_mtime` zawiera czas ostatniej modyfikacji pliku, to znaczy czas ostatniej operacji zapisu do niego.
- Pole `st_ctime` zawiera czas ostatniej zmiany statusu pliku. Jest to często mylone z czasem utworzenia pliku, który nie jest zachowywany w Linuksie ani w innych systemach uniksowych. Pole to w rzeczywistości opisuje czas ostatniej modyfikacji metadanych dla pliku (na przykład, nazwy jego właściciela lub uprawnień dostępu).

W przypadku sukcesu wszystkie trzy funkcje zwracają wartość zero oraz zapamiętują wartości metadanych dla pliku w dostarczonej strukturze `stat`. W przypadku błędu zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnień wyszukiwania w jednym z elementów ścieżki `path` (dotyczy wyłącznie `stat()` oraz `lstat()`).

`EBADF`
: Deskryptor pliku `fd` jest niepoprawny (dotyczy tylko `fstat()`).

`EFAULT`
: Parametry `path` lub `buf` są niepoprawnymi wskaźnikami.

`ELOOP`
: Ścieżka `path` zawiera zbyt dużo dowiązań symbolicznych (dotyczy tylko `stat()` oraz `lstat()`).

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa (dotyczy tylko `stat()` oraz `lstat()`).

`ENOENT`
: Element ścieżki `path` nie istnieje (dotyczy tylko `stat()` oraz `lstat()`).

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOTDIR`
: Element w ścieżce `path` nie jest katalogiem (dotyczy tylko `stat()` oraz `lstat()`).

Następujący program używa funkcji `stat()`, aby odczytać rozmiar pliku, którego nazwa dostarczona jest w linii poleceń:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
   struct stat sb;
   int ret;

   if (argc < 2)
```

```
    {
       fprintf (stderr, "Użycie programu: %s <nazwa pliku>\n", argv[0]);
       return 1;
    }

    ret = stat (argv[1], &sb);
    if (ret)
    {
       perror ("stat");
       return 1;
    }

    printf ("%s ma rozmiar %ld bajtów\n", argv[1], sb.st_size);

    return 0;
}
```

Oto wynik uruchomienia programu z parametrem, równym nazwie jego własnego pliku źródłowego:

```
$ ./stat stat.c
stat.c ma rozmiar 440 bajtów
```

Z kolei poniższy fragment kodu używa funkcji `fstat()`, aby sprawdzić, czy otwarty już plik jest urządzeniem fizycznym (w przeciwieństwie do urządzenia sieciowego):

```
/*
* is_on_physical_device – funkcja zwraca liczbę dodatnią,
* jeśli deskryptor pliku 'fd' znajduje się w urządzeniu fizycznym,
* natomiast 0, jeśli plik znajduje się w niefizycznym
* lub wirtualnym urządzeniu (np. w zamontowanym systemie plików NFS).
* Funkcja zwraca -1 w przypadku błędu.
*/
int is_on_physical_device (int fd)
{
   struct stat sb;
   int ret;

   ret = fstat (fd, &sb);
   if (ret)
   {
      perror ("fstat");
      return -1;
   }

   return gnu_dev_major (sb.st_dev);
}
```

Uprawnienia

Podczas gdy funkcje z grupy *stat* mogą zostać użyte w celu odczytania uprawnień istniejących dla danego pliku, inne funkcje systemowe służą do ustawiania tych wartości:

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

int chmod (const char *path, mode_t mode);
int fchmod (int fd, mode_t mode);
```

Obie funkcje `chmod()` oraz `fchmod()` ustawiają uprawnienia pliku na wartość przekazaną w parametrze `mode`. W przypadku funkcji `chmod()`, parametr `path` oznacza względną lub bezwzględną ścieżkę do pliku, który powinien zostać zmodyfikowany. W przypadku funkcji `fchmod()`, plik reprezentowany jest przez deskryptor pliku `fd`.

Poprawne wartości parametru `mode`, który reprezentowany jest przez nieprzezroczysty typ całkowitoliczbowy `mode_t`, są takie same jak w przypadku wartości otrzymanych w polu `st_mode` struktury `stat`. Chociaż wartości te są zwykłymi liczbami całkowitymi, ich znaczenie jest specyficzne dla każdej implementacji Uniksa. Zgodnie z tym POSIX definiuje zestaw stałych, które reprezentują różne uprawnienia (szczegóły znajdują się w podrozdziale zatytułowanym Uprawnienia nowych plików, znajdującym się w rozdziale 2.). Stałe te mogą być łączone ze sobą za pomocą operatora sumy binarnej, tworząc poprawne wartości parametru `mode`. Na przykład zestaw stałych `(S_IRUSR | S_IRGRP)` ustawia uprawnienia w pliku do odczytu przez użytkownika i grupę właścicielską.

Aby zmienić uprawnienia pliku, efektywny identyfikator procesu, który wywołuje funkcje `chmod()` lub `fchmod()`, musi być zgodny z właścicielem pliku lub proces musi posiadać uprawnienia `CAP_FOWNER`.

W przypadku sukcesu funkcje zwracają wartość zero. W przypadku błędu zwracają `−1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnień szukania dla elementu ścieżki `path` (tylko dla `chmod()`).

`EBADF`
: Deskryptor pliku `fd` jest nieprawidłowy (tylko dla `fchmod()`).

`EFAULT`
: Parametr `path` jest błędnym wskaźnikiem (tylko dla `chmod()`).

`EIO`
: W systemie plików wystąpił wewnętrzny błąd operacji wejścia i wyjścia. Jest to niepokojący błąd, który może oznaczać, że uszkodzony został dysk lub system plików.

`ELOOP`
: Jądro napotkało zbyt wiele dowiązań symbolicznych podczas analizowania ścieżki `path` (tylko dla `chmod()`).

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa (tylko dla `chmod()`).

`ENOENT`
: Ścieżka `path` nie istnieje (tylko dla `chmod()`).

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOTDIR`
: Element w ścieżce `path` nie jest katalogiem (tylko dla `chmod()`).

`EPERM`
: Efektywny identyfikator procesu wywołującego nie pasuje do właściciela pliku, a sam proces nie posiada uprawnienia `CAP_FOWNER`.

`EROFS`
: Plik znajduje się w systemie plików z uprawnieniami tylko do odczytu.

Poniższy fragment kodu ustawia uprawnienia dla pliku *map.png*, pozwalające na wykonywanie operacji odczytu i zapisu przez jego właściciela:

```
int ret;
/*
* Ustawia uprawnienia dla pliku 'map.png' znajdującego się w aktualnym katalogu,
* które pozwalają właścicielowi pliku na wykonywanie jego odczytu i zapisu.
* Jest to równoważne poleceniu 'chmod 600 ./map.png'.
*/
ret = chmod ("./map.png", S_IRUSR | S_IWUSR);
if (ret)
   perror ("chmod");
```

Poniższy fragment kodu wykonuje tę samą operację jak poprzedni, przy założeniu, że parametr `fd` oznacza otwarty plik *map.png*:

```
int ret;
/*
* Ustawia uprawnienia do pliku wskazywanego prze 'fd',
* pozwalające na jego odczyt i zapis przez właściciela.
*/
ret = fchmod (fd, S_IRUSR | S_IWUSR);
if (ret)
   perror ("fchmod");
```

Obie funkcje `chmod()` oraz `fchmod()` dostępne są we wszystkich nowoczesnych systemach uniksowych. POSIX wymaga istnienia pierwszej z nich, natomiast druga jest opcjonalna.

Prawa własności

W strukturze `stat`, pola `st_uid` oraz `st_gid` dostarczają informacji opisujących odpowiednio właściciela i grupę właścicielską dla pliku. Użytkownik może zmienić te dwie wartości za pomocą trzech funkcji systemowych:

```
#include <sys/types.h>
#include <unistd.h>

int chown (const char *path, uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
```

Funkcje `chown()` oraz `lchown()` ustawiają prawa własności dla pliku wskazywanego przez ścieżkę `path`. Wyniki ich działania są takie same, chyba że plik jest dowiązaniem symbolicznym: pierwsza z tych funkcji używa informacji zawartej w dowiązaniu symbolicznym, aby uzyskać dostęp do właściwego pliku i ustawić dla niego odpowiednie prawa własności. Natomiast funkcja `lchown()`zachowuje się przeciwnie — zmienia prawa własności tylko dla samego dowiązania symbolicznego. Funkcja `fchown()` ustawia prawa własności dla pliku, reprezentowanego przez deskryptor pliku `fd`.

W przypadku sukcesu, wszystkie trzy funkcje ustawiają właściciela pliku na wartość przekazaną w parametrze `owner`, grupę właścicielską pliku na wartość przekazaną w parametrze `group` oraz zwracają zero. Jeśli któryś z parametrów `owner` lub `group` równy jest `-1`, wówczas ta wartość nie zostaje wzięta pod uwagę. Tylko proces z uprawnieniem `CAP_CHOWN` (zwykle proces administratora) może zmienić właściciela pliku. Właściciel pliku może zmienić grupę właścicielską pliku na dowolną grupę, do której należy; procesy z uprawnieniem `CAP_CHOWN` mogą zmienić grupę na dowolną wartość.

W przypadku błędu, funkcje zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

EACCESS
: Proces wywołujący nie posiada uprawnień szukania dla elementu ścieżki `path` (tylko dla `chown()`oraz `lchown()`).

EBADF
: Deskryptor pliku `fd` jest nieprawidłowy (tylko dla `fchown()`).

EFAULT
: Parametr `path` jest niepoprawny (tylko dla `chown()`oraz `lchown()`).

EIO
: Wystąpił wewnętrzny błąd operacji wejścia i wyjścia (jest to poważny problem).

ELOOP
: Jądro napotkało zbyt wiele dowiązań symbolicznych podczas analizowania ścieżki `path` (tylko dla `chown()`oraz `lchown()`).

ENAMETOOLONG
: Ścieżka `path` jest zbyt długa (tylko dla `chown()`oraz `lchown()`).

ENOENT
: Plik nie istnieje.

ENOMEM
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

ENOTDIR
: Element w ścieżce `path` nie jest katalogiem (tylko dla `chown()`oraz `lchown()`).

EPERM
: Proces wywołujący nie posiada wymaganych uprawnień, aby zmienić właściciela lub grupę właścicielską.

EROFS
: System plików posiada uprawnienia dostępu tylko do odczytu.

Poniższy fragment kodu zmienia grupę właścicielską dla pliku *manifest.txt*, znajdującego się w aktualnym katalogu roboczym, na wartość *officers*. Aby operacja zakończyła się sukcesem, wywołujący użytkownik powinien posiadać uprawnienie `CAP_CHOWN` lub mieć nazwę *kidd* i należeć do grupy *officers*:

```
struct group *gr;
int ret;

/*
* Funkcja getgrnam( ) zwraca informacje o grupie,
* której nazwa przekazana musi zostać w parametrze.
*/
gr = getgrnam ("officers");
if (!gr)
{
   /* prawdopodobnie niewłaściwa grupa */
   perror ("getgrnam");
   return 1;
}

/* ustawia grupę właścicielską dla pliku manifest.txt na wartość 'officers' */
ret = chmod ("manifest.txt", -1, gr->gr_gid);
if (ret)
   perror ("chmod");
```

Przed wywołaniem programu grupą właścicielską pliku był *crew*:

```
$ ls -l
-rw-r--r-- 1 kidd crew 13274 May 23 09:20 manifest.txt
```

Po uruchomieniu programu uprawnienia grupy właścicielskiej dla pliku zostały przekazane do *officers*:

```
$ ls -l
-rw-r--r-- 1 kidd officers 13274 May 23 09:20 manifest.txt
```

Właściciel pliku o nazwie *kidd* nie został zmieniony, gdyż w powyższym fragmencie kodu przekazano wartość `-1` w parametrze `uid`.

Poniższa funkcja ustawia właściciela oraz grupę właścicielską dla pliku, który reprezentowany jest przez deskryptor pliku `fd`, na administratora (*root*):

```
/*
* make_root_owner - zmienia właściciela i grupę właścicielską pliku,
* reprezentowanego przez 'fd', na administratora.
* Zwraca 0 w przypadku sukcesu lub -1 w przypadku niepowodzenia.
*/
int make_root_owner (int fd)
{
   int ret;

   /* w przypadku administratora zarówno GID, jak również UID wynosi 0 */
   ret = fchown (fd, 0, 0);
   if (ret)
      perror ("fchown");

   return ret;
}
```

Proces wywołujący musi posiadać uprawnienie `CAP_CHOWN`. Zazwyczaj jego właścicielem powinien być administrator.

Atrybuty rozszerzone

Atrybuty rozszerzone, zwane także *xattrs*, dostarczają mechanizmu pozwalającego na stałe wiązać parametry klucza i wartości z plikami. W tym rozdziale omówiono już wszystkie rodzaje metadanych typu „klucz-wartość", które związane są z plikami, jak rozmiar pliku, nazwa właściciela, data ostatniej modyfikacji itd. Atrybuty rozszerzone pozwalają w przypadku istniejących systemów plików na obsługę nowych opcji, które nie były przewidziane podczas ich projektowania, takich jak narzucona kontrola dostępu dla celów bezpieczeństwa. Interesującą cechą w przypadku atrybutów rozszerzonych jest to, że aplikacje z przestrzeni użytkownika mogą dowolnie tworzyć, odczytywać i zapisywać metadane typu „klucz-wartość".

Atrybuty rozszerzone *nie są związane z systemem plików* — oznacza to, że aplikacje używają standardowego interfejsu, aby je modyfikować; ten interfejs nie jest specyficzny dla żadnego systemu plików. Aplikacje mogą więc używać atrybutów rozszerzonych bez obawy o to, w jakim systemie plików znajdują się dane pliki lub w jaki sposób system plików przechowuje wewnętrznie klucze i wartości. Pomimo tego implementacja atrybutów rozszerzonych jest wciąż zależna od systemu plików. Różne systemy plików przechowują atrybuty rozszerzone w odmienny sposób, lecz jądro ukrywa te szczegóły implementacji, tworząc abstrakcję dzięki udostępnieniu interfejsu atrybutów rozszerzonych.

Na przykład, system plików *ext3* przechowuje atrybuty rozszerzone dla pliku w niewykorzystanym obszarze jego i-węzła[1]. Opcja ta pozwala na bardzo szybkie odczytywanie atrybutów rozszerzonych. Ponieważ blok systemu plików, który zawiera i-węzeł, zostaje zawsze odczytany z dysku do pamięci, gdy aplikacja przeprowadza operację dostępu do pliku, dlatego też atrybuty rozszerzone „automatycznie" zostaną również odczytane do pamięci i mogą być dostępne bez żadnych dodatkowych kosztów.

Inne systemy plików, jak *FAT* czy *minixfs*, w ogóle nie wspierają atrybutów rozszerzonych. Zwracają kod błędu `ENOTSUP`, gdy następuje próba wykonania operacji dotyczącej atrybutów rozszerzonych dla ich plików.

Klucze i wartości

Każdy atrybut rozszerzony identyfikowany jest przez unikalny *klucz*. Klucze muszą być poprawnie zapisane w standardzie kodowania znaków UTF-8. Format ich zapisu jest następujący: *przestrzeń_ nazw.atrybut*. Każdy klucz musi być pełną nazwą, to znaczy powinien zaczynać się od ciągu znaków, opisującego poprawną przestrzeń nazw, po którym następuje kropka. Przykładem poprawnego klucza może być ciąg znaków *user.mime_type*; klucz ten należy do przestrzeni nazw *user*, a posiada atrybut *mime_type*.

Klucz może być *zdefiniowany* lub *niezdefiniowany*. Jeśli klucz jest zdefiniowany, jego wartość może być pusta lub niepusta. Oznacza to, że istnieje różnica między kluczem niezdefiniowanym, a kluczem zdefiniowanym, nieposiadającym przypisanej mu wartości. Wymagane jest istnienie specjalnego interfejsu, który pozwoli na usuwanie kluczy (niewystarczające jest przypisywanie im pustych wartości).

Wartość niepusta, związana z kluczem, może być dowolną tablicą bajtów. Ponieważ wartość ta niekoniecznie musi być ciągiem znaków, dlatego też nie jest wymagane, aby była zakończona znakiem pustym, choć takie zakończenie na pewno ma sens w przypadku, gdy należy przechować w niej łańcuch znaków dla języka C. Ponieważ nie ma gwarancji, że wartości będą zakończone znakiem pustym, dlatego też wszystkie operacje dotyczące atrybutów rozszerzonych wymagają podania rozmiaru pola wartości. Podczas odczytywania atrybutu, rozmiar zostaje dostarczony przez jądro; podczas zapisywania atrybutu, użytkownik musi dostarczyć wartość rozmiaru.

Linux nie narzuca żadnych ograniczeń dotyczących liczby kluczy, długości klucza, rozmiaru wartości czy całkowitej przestrzeni, która może być zużyta dla przechowywania wszystkich kluczy i wartości powiązanych z danym plikiem. Systemy plików posiadają jednak swoje własne realne limity. Są one zazwyczaj widoczne jako ograniczenia całkowitego rozmiaru wszystkich kluczy oraz wartości związanych z danym plikiem.

Na przykład, w *ext3* wszystkie atrybuty rozszerzone dla danego pliku muszą mieścić się w niezajętym obszarze, przeznaczonym na jego i-węzeł oraz zajmować maksymalnie jeden dodatkowy blok systemu plików (starsze wersje systemu plików *ext3* mogły używać do tego celu tylko jednego bloku, natomiast nie miały dostępu do obszaru i-węzła). Wynika stąd rzeczywiste ograniczenie, zawierające się pomiędzy 1 kB a 8 kB dla pojedynczego pliku, w zależności od

[1] Oczywiście dopóki wystarczy miejsca dla i-węzła. System plików *ext3* przechowuje atrybuty rozszerzone w dodatkowych blokach systemu plików. Starsze wersje *ext3* nie posiadały możliwości przechowywania atrybutów rozszerzonych wewnątrz i-węzła.

Lepszy sposób przechowywania typów MIME

Zarządcy plików z graficznym interfejsem użytkownika, takie jak Nautilus, działający w środowisku GNOME, zachowują się w odmienny sposób w zależności od różnych typów plików: dostarczają unikalne ikony, odmienne zachowanie podczas naciskania klawiszy myszki, specjalną listę operacji do wykonania itd. Aby to uzyskać, zarządca musi znać format każdego pliku. By ustalić jego format, w systemach plików, takich jak Windows, używana jest prosta metoda sprawdzania rozszerzenia nazwy pliku. Z powodów związanych zarówno z tradycją, jak i bezpieczeństwem, w systemach uniksowych stosuje się metodę sprawdzania pliku i dopiero na podstawie tego interpretuje się jego typ. Proces ten zwany jest *rozpoznawaniem typu MIME* (ang. *MIME type sniffing*).

Niektórzy zarządcy plików generują tę informację w locie; inni tworzą ją jednokrotnie i następnie buforują, często umieszczając ją we własnej bazie danych. Zarządca plików musi ciągle synchronizować tę bazę danych z plikami, które mogą się zmieniać bez jego wiedzy. Lepszym rozwiązaniem tego problemu jest pozbycie się własnej bazy danych i przechowywanie takich metadanych w atrybutach rozszerzonych: można nimi wówczas łatwiej zarządzać, istnieje do nich szybszy dostęp i są łatwiej obsługiwane przez każdą aplikację.

rozmiaru bloków w systemie plików. W przeciwieństwie do *ext3*, system plików *XFS* nie posiada rzeczywistych ograniczeń. Nawet w przypadku systemu plików *ext3* te ograniczenia nie są jednak problemem, ponieważ większość kluczy i wartości jest zapisywanych w postaci krótkich łańcuchów tekstowych. Mimo tego należy pamiętać, aby wcześniej dwa razy się zastanowić, zanim umieści się całą historię poprawek dla projektu w rozszerzonych atrybutach pliku!

Przestrzenie nazw dla atrybutów rozszerzonych

Przestrzenie nazw, związane z atrybutami rozszerzonymi, są czymś więcej niż tylko narzędziami organizacyjnymi. Jądro narzuca różne strategie dostępu, w zależności od danej przestrzeni nazw.

Aktualnie w Linuksie zdefiniowane są cztery przestrzenie nazw dla atrybutów rozszerzonych, a w przyszłości może pojawić się ich więcej. Oto cztery aktualne:

system
: Przestrzeń nazw *system* używana jest w celu zaimplementowania opcji jądra, służących do wykorzystania atrybutów rozszerzonych, takich jak listy kontroli dostępu (ACL). Przykładem atrybutu rozszerzonego w tej przestrzeni nazw jest *system.posix_acl_access*. Od aktywnego modułu bezpieczeństwa zależy, czy użytkownicy mogą czytać lub zapisywać te atrybuty. Należy założyć najgorszy przypadek, że żaden użytkownik (włączając administratora) nie może nawet odczytać tych atrybutów.

security
: Przestrzeń nazw *security* używana jest w celu zaimplementowania modułów bezpieczeństwa, takich jak SELinux. Ponownie od aktywnego modułu bezpieczeństwa zależy, czy aplikacje z przestrzeni użytkownika będą mogły uzyskać dostęp do atrybutów z tej przestrzeni nazw. Domyślnie wszystkie procesy mogą odczytywać te atrybuty, lecz tylko procesy z uprawnieniem `CAP_SYS_ADMIN` mogą je zapisywać.

trusted

Przestrzeń nazw *trusted* przechowuje zastrzeżone informacje w przestrzeni użytkownika. Tylko procesy z uprawieniem `CAP_SYS_ADMIN` mogą czytać lub zapisywać te atrybuty.

user

Przestrzeń nazw *user* jest standardowo używana przez zwykłe procesy. Jądro zarządza dostępem do tej przestrzeni nazw poprzez użycie zwykłych bitów uprawnień pliku. Aby odczytać wartość dla istniejącego klucza, proces musi posiadać uprawnienia odczytu do danego pliku. By stworzyć nowy klucz lub zapisać wartość do istniejącego klucza, proces musi posiadać uprawnienia zapisu do danego pliku. W tej przestrzeni nazw atrybuty rozszerzone można przypisać jedynie do plików zwykłych, natomiast nie da się tego zrealizować w przypadku dowiązań symbolicznych lub plików urządzeń. Jest to zalecana przestrzeń nazw podczas projektowania aplikacji z przestrzeni użytkownika, która zamierza używać atrybutów rozszerzonych.

Operacje dla atrybutów rozszerzonych

POSIX definiuje cztery rodzaje operacji, które mogą zostać przeprowadzone przez aplikacje dla atrybutów rozszerzonych w danym pliku:

- Po podaniu pliku i klucza nastąpi zwrócenie odpowiedniej wartości.
- Po podaniu pliku, klucza i wartości nastąpi przypisanie tejże wartości do klucza.
- Po podaniu pliku nastąpi zwrócenie listy wszystkich kluczy atrybutów rozszerzonych dla tego pliku.
- Po podaniu pliku i klucza nastąpi usunięcie atrybutu rozszerzonego z pliku.

Dla każdej z tych operacji POSIX dostarcza trzy rodzaje funkcji systemowych:

- Wersję, która działa dla podanej ścieżki do pliku; jeśli ścieżka wskazuje na dowiązanie symboliczne, operacja dotyczy pliku wskazywanego przez to dowiązanie (jest to zachowanie standardowe).
- Wersję, która działa dla podanej ścieżki do pliku; jeśli ścieżka wskazuje na dowiązanie symboliczne, operacja dotyczy samego dowiązania (jest to wersja `l` funkcji systemowej).
- Wersję, która działa dla deskryptora pliku (jest to wersja `f` standardu).

W następnych podrozdziałach zostaną omówione wszystkie (12) permutacje powyższych opcji.

Odczytywanie atrybutu rozszerzonego. Najprostszą operacją jest odczytywanie wartości atrybutu rozszerzonego z pliku, po podaniu określonego klucza:

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t getxattr (const char *path, const char *key, void *value, size_t size);
ssize_t lgetxattr (const char *path, const char *key, void *value, size_t size);
ssize_t fgetxattr (int fd, const char *key, void *value, size_t size);
```

Poprawne wywołanie funkcji `getxattr()` zapisuje atrybut rozszerzony o nazwie `key` dla pliku `path`, w dostarczonym buforze `value` o rozmiarze `size`. Rozmiar podawany jest w bajtach. Funkcja zwraca rzeczywisty rozmiar wartości.

Jeśli parametr `size` równy jest zeru, wywołanie funkcji zwraca tylko rozmiar wartości, nie zapisując jej w parametrze `value`. Dlatego też przekazanie `0` w tym parametrze pozwala aplikacjom

na ustalenie poprawnego rozmiaru bufora, w którym powinna zostać przechowana wartość klucza. Po otrzymaniu tego rozmiaru aplikacje mogą następnie w razie potrzeby przydzielić lub zmienić obszar pamięci dla bufora.

Funkcja `lgetxattr()` działa prawie tak samo jak `getxattr()`, za wyjątkiem tego, że jeśli ścieżka `path` jest dowiązaniem symbolicznym, wówczas zwrócone zostają atrybuty rozszerzone dla samego dowiązania, a nie pliku, wskazywanego przez tę ścieżkę. Z poprzedniego podrozdziału wynika, że atrybuty w przestrzeni użytkownika nie mogą być używane w przypadku dowiązań symbolicznych, stąd też ta funkcja jest rzadko używana.

Funkcja `fgetxattr()` używa deskryptora pliku `fd`; poza tym działa tak samo jak funkcja `getxattr()`.

W przypadku błędu wszystkie trzy funkcje zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnienia przeszukiwania dla katalogu będącego jednym z elementów ścieżki `path` (tylko w przypadku `getxattr()` oraz `lgetxattr()`).

`EBADF`
: Deskryptor pliku `fd` jest niepoprawny (tylko w przypadku `fgetxattr()`).

`EFAULT`
: Parametry `path`, `key` lub `value` są niepoprawnymi wskaźnikami.

`ELOOP`
: Parametr `path` zawiera zbyt dużo dowiązań symbolicznych (tylko w przypadku `getxattr()` oraz `lgetxattr()`).

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa (tylko w przypadku `getxattr()` oraz `lgetxattr()`).

`ENOATTR`
: Parametr `key` zawiera nazwę klucza atrybutu, który nie istnieje lub proces nie posiada dostępu do atrybutu.

`ENOENT`
: Element ścieżki `path` nie istnieje (tylko w przypadku `getxattr()` oraz `lgetxattr()`).

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOTDIR`
: Element ścieżki `path` nie jest katalogiem (tylko w przypadku `getxattr()` oraz `lgetxattr()`).

`ENOTSUP`
: System plików, w którym istnieje plik, wskazywany przez parametr `path` lub `fd`, nie wspiera atrybutów rozszerzonych.

`ERANGE`
: Rozmiar `size` jest zbyt mały, aby przechować wartość klucza. Jak zostało wcześniej napisane, funkcja ta może zostać ponownie wywołana z parametrem `size` równym zeru; wartość zwrócona wyznaczy wówczas wymagany rozmiar bufora i parametr `size` będzie mógł być odpowiednio poprawiony.

Ustawianie atrybutu rozszerzonego. Następujące trzy funkcje systemowe ustawiają dany atrybut rozszerzony:

```
#include <sys/types.h>
#include <attr/xattr.h>

int setxattr (const char *path, const char *key, const void *value, size_t size,
  int flags);
int lsetxattr (const char *path, const char *key, const void *value, size_t size,
  int flags);
int fsetxattr (int fd, const char *key, const void *value, size_t size, int flags);
```

Poprawne wywołanie funkcji `setxattr()` ustawia dla pliku `path` klucz atrybutu rozszerzonego na wartość `value`, której rozmiar wynosi `size` i jest podawany w bajtach. Pole `flags` pozwala na modyfikowanie zachowania funkcji. Jeśli `flags` równy jest `XATTR_CREATE`, wywołanie funkcji nie powiedzie się, gdy dany atrybut rozszerzony już istnieje. Jeśli `flags` równy jest `XATTR_` ↪`REPLACE`, wywołanie funkcji nie powiedzie się, gdy dany atrybut rozszerzony nie istnieje. Domyślny sposób działania — aktywny, gdy parametr `flags` równy jest zeru — pozwala zarówno na utworzenie, jak i zmianę atrybutu. Bez względu na wartość parametru `flags`, klucze o wartościach innych niż parametr `key`, nie są brane pod uwagę przez funkcję `setxattr()`.

Funkcja `lsetxattr()` działa prawie tak samo jak `setxattr()`, za wyjątkiem tego, że jeśli ścieżka `path` jest dowiązaniem symbolicznym, wówczas zostają ustawione atrybuty rozszerzone dla samego dowiązania, zamiast dla pliku wskazywanego przez tę ścieżkę. Jak już napisano, atrybuty w przestrzeni użytkownika nie mogą być używane w przypadku dowiązań symbolicznych, stąd też ta funkcja jest również rzadko używana.

Funkcja `fsetxattr()` używa deskryptora pliku `fd`; poza tym, działa tak samo jak funkcja `set` ↪`xattr()`.

W przypadku sukcesu wszystkie trzy funkcje zwracają `0`; w przeciwnym razie zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnienia przeszukiwania dla katalogu będącego jednym z elementów ścieżki `path` (tylko w przypadku `setxattr()` oraz `lsetxattr()`).

`EBADF`
: Deskryptor pliku `fd` jest niepoprawny (tylko w przypadku `fsetxattr()`).

`EDQUOT`
: Limit miejsca na dysku dla użytkownika (*quota*) nie pozwala na wykonanie żądanej operacji.

`EEXIST`
: W parametrze `flags` została ustawiona wartość `XATTR_CREATE`, a dla danego pliku istnieje już klucz `key`.

`EFAULT`
: Parametry `path`, `key` lub `value` są niepoprawnymi wskaźnikami.

`EINVAL`
: Parametr `flags` jest niepoprawny.

`ELOOP`
: Parametr `path` zawiera zbyt wiele dowiązań symbolicznych (tylko w przypadku `set` ↪`xattr()` oraz `lsetxattr()`).

ENAMETOOLONG
: Ścieżka `path` jest zbyt długa (tylko w przypadku `setxattr()` oraz `lsetxattr()`).

ENOATTR
: W parametrze `flags` została ustawiona wartość `XATTR_REPLACE`, a dla danego pliku nie istnieje klucz, przekazany w parametrze `key`.

ENOENT
: Element ścieżki `path` nie istnieje (tylko w przypadku `setxattr()` oraz `lsetxattr()`).

ENOMEM
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

ENOSPC
: W systemie plików nie ma wystarczająco miejsca, aby zapisać atrybut rozszerzony.

ENOTDIR
: Element ścieżki `path` nie jest katalogiem (tylko w przypadku `setxattr()` oraz `lset` ↪`xattr()`).

ENOTSUP
: System plików, w którym istnieje plik, wskazywany przez parametr `path` lub `fd`, nie wspiera atrybutów rozszerzonych.

Lista atrybutów rozszerzonych dla pliku. Następujące trzy funkcje systemowe wyprowadzają zestaw kluczy atrybutów rozszerzonych, przypisanych do danego pliku:

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t listxattr (const char *path, char *list, size_t size);
ssize_t llistxattr (const char *path, char *list, size_t size);
ssize_t flistxattr (int fd, char *list, size_t size);
```

Poprawne wywołanie funkcji `listxattr()` zwraca listę kluczy atrybutów przypisanych do pliku, określonego w parametrze `path`. Lista zapisana zostaje w buforze, dostarczonym w parametrze `list`, który posiada rozmiar `size`, określony w bajtach. Funkcja systemowa zwraca rzeczywisty rozmiar listy, również wyrażony w bajtach.

Każdy klucz atrybutu rozszerzonego, zwrócony w liście `list`, zakończony jest znakiem pustym, więc przykładowa lista może wyglądać następująco:

```
"user.md5_sum\0user.mime_type\0system.posix_acl_default\0"
```

Dlatego też, mimo że każdy klucz jest tradycyjnym łańcuchem znakowym języka C, zakończonym znakiem pustym, wymagana jest jednak znajomość długości całej listy (którą można uzyskać z wartości powrotnej funkcji), aby móc ją całą przetworzyć. By dowiedzieć się, ile pamięci trzeba przydzielić na bufor, należy wywołać jedną z powyższych funkcji z parametrem `size` równym zero; spowoduje to, że funkcja zwróci rzeczywistą długość pełnej listy kluczy. Podobnie jak w przypadku funkcji `getxattr()` aplikacje mogą wykorzystać tę funkcjonalność, aby przydzielić lub zmienić obszar pamięci dla bufora, przekazując odpowiednią wartość w parametrze `size`.

Funkcja `llistxattr()` działa prawie tak samo jak `listxattr()`, za wyjątkiem tego, że jeśli ścieżka `path` jest dowiązaniem symbolicznym, wówczas zostanie wygenerowana lista kluczy atrybutów rozszerzonych dla samego dowiązania, zamiast dla pliku wskazywanego przez tę

ścieżkę. Jak już napisano, atrybuty w przestrzeni użytkownika nie mogą być używane w przypadku dowiązań symbolicznych, stąd też ta funkcja jest rzadko używana.

Funkcja `flistxattr()` używa deskryptora pliku `fd`; poza tym działa tak samo jak funkcja `listxattr()`.

W przypadku błędu wszystkie trzy funkcje zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnienia przeszukiwania dla katalogu będącego jednym z elementów ścieżki `path` (tylko w przypadku `listxattr()` oraz `llistxattr()`).

`EBADF`
: Deskryptor pliku `fd` jest niepoprawny (tylko w przypadku `flistxattr()`).

`EFAULT`
: Parametry `path` lub `list` są niepoprawnymi wskaźnikami.

`ELOOP`
: Parametr `path` zawiera zbyt dużo dowiązań symbolicznych (tylko w przypadku `list↪xattr()` oraz `llistxattr()`).

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa (tylko w przypadku `listxattr()` oraz `llistxattr()`).

`ENOENT`
: Element ścieżki `path` nie istnieje (tylko w przypadku `listxattr()` oraz `llistxattr()`).

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOTDIR`
: Element ścieżki `path` nie jest katalogiem (tylko w przypadku `listxattr()` oraz `llist↪xattr()`).

`ENOTSUP`
: System plików, w którym istnieje plik, wskazywany przez parametr `path` lub `fd`, nie wspiera atrybutów rozszerzonych.

`ERANGE`
: Rozmiar `size` jest niezerowy i zbyt mały, aby przechować pełną listę kluczy. Funkcja ta może ponownie zostać wywołana z parametrem `size` równym zeru, aby wyznaczyć rzeczywisty rozmiar listy. W programie można następnie zmienić parametr `size` dla bufora i ponownie wywołać funkcję.

Usuwanie atrybutu rozszerzonego. Wreszcie, poniższe trzy funkcje systemowe pozwalają na usunięcie danego klucza z pliku:

```
#include <sys/types.h>
#include <attr/xattr.h>

int removexattr (const char *path, const char *key);
int lremovexattr (const char *path, const char *key);
int fremovexattr (int fd, const char *key);
```

Poprawne wywołanie funkcji `removexattr()` usuwa klucz atrybutu rozszerzonego z pliku `path`. Należy zwrócić uwagę (jak wcześniej napisano), że istnieje różnica pomiędzy kluczem niezdefiniowanym a kluczem zdefiniowanym, posiadającym wartość pustą (o długości zero).

Funkcja `lremovexattr()` działa prawie tak samo jak `removexattr()`, za wyjątkiem tego, że jeśli ścieżka `path` jest dowiązaniem symbolicznym, wówczas zostanie usunięty klucz atrybutu rozszerzonego dla samego dowiązania, zamiast dla pliku wskazywanego przez tę ścieżkę. Jak już napisano, atrybuty w przestrzeni użytkownika nie mogą być używane w przypadku dowiązań symbolicznych, stąd też ta funkcja jest również rzadko używana.

Funkcja `fremovexattr()` używa deskryptora pliku `fd`; poza tym, działa tak samo jak funkcja `removexattr()`.

W przypadku sukcesu wszystkie trzy funkcje zwracają `0`. W przypadku błędu zwracają `-1` oraz odpowiednio ustawiają zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnienia przeszukiwania dla katalogu będącego jednym z elementów ścieżki `path` (tylko w przypadku `removexattr()` oraz `lremovexattr()`).

`EBADF`
: Deskryptor pliku `fd` jest niepoprawny (tylko w przypadku `fremovexattr()`).

`EFAULT`
: Parametry `path` lub `key` są niepoprawnymi wskaźnikami.

`ELOOP`
: Parametr `path` zawiera zbyt wiele dowiązań symbolicznych (tylko w przypadku `remove` ↪`xattr()` oraz `lremovexattr()`).

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa (tylko w przypadku `removexattr()` oraz `lremovexattr()`).

`ENOATTR`
: Klucz, podany w parametrze `key`, nie istnieje dla danego pliku.

`ENOENT`
: Element ścieżki `path` nie istnieje (tylko w przypadku `removexattr()` oraz `lremovexattr()`).

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOTDIR`
: Element ścieżki `path` nie jest katalogiem (tylko w przypadku `removexattr()` oraz `lremove` ↪`xattr()`).

`ENOTSUP`
: System plików, w którym istnieje plik, wskazywany przez parametr `path` lub `fd`, nie wspiera atrybutów rozszerzonych.

Katalogi

Katalog (ang. *directory*) jest prostą koncepcją w systemie Unix: zawiera listę nazw plików, z których każdy odwzorowuje numer i-węzła. Każda nazwa zwana jest *elementem katalogu* (ang. *directory entry*), a każde odwzorowanie nazwy na i-węzeł jest *dowiązaniem*. Zawartość katalogu —

to, co jest widoczne dla użytkownika po wykonaniu polecenia *ls* — jest listą znajdujących się w nim wszystkich nazw plików. Gdy użytkownik otwiera plik w danym katalogu, jądro szuka nazwy tego pliku w liście, aby znaleźć odpowiadający mu numer i-węzła. Następnie jądro przekazuje ten numer i-węzła do systemu plików, który używa go w celu ustalenia fizycznego położenia pliku w urządzeniu.

Katalogi mogą zawierać również inne katalogi. *Podkatalog* (ang. *subdirectory*) to katalog istniejący wewnątrz innego katalogu. Opierając się na tej definicji, wszystkie katalogi są podkatalogami należącymi do pewnego *katalogu nadrzędnego* (ang. *parent directory*), za wyjątkiem katalogu, znajdującego się w korzeniu drzewa katalogów systemu plików, oznaczanego ukośnikiem: /. Katalog taki zwany jest *katalogiem głównym* (ang. *root directory*) — nie należy mylić go z katalogiem własnym użytkownika administracyjnego: */root*.

Ścieżka (ang. *pathname*) zawiera nazwę pliku wraz z jednym lub więcej jego katalogami nadrzędnymi. *Ścieżka bezwzględna* (ang. *absolute pathname*) zaczyna się od katalogu głównego — na przykład */usr/bin/sextant*. Ścieżka względna (ang. *relative pathname*), taka jak *bin/sextant*, nie zaczyna się od katalogu głównego. Aby była przydatna, system operacyjny musi znać katalog, dla którego ścieżka ta jest względna. Aktualny katalog roboczy (omawiany w następnym podrozdziale) jest używany jako punkt początkowy.

Nazwy plików i katalogów mogą składać się z dowolnych znaków za wyjątkiem ukośnika /, który rozdziela katalogi w ścieżce, a także za wyjątkiem *znaku pustego*, kończącego nazwę ścieżki. Działaniem standardowym jest ograniczanie znaków używanych w nazwach ścieżek do poprawnych znaków drukowalnych, a także znaków zawierających się w zbiorze znaków regionalnych lub nawet znaków ASCII. Ponieważ ani jądro, ani biblioteka języka C nie narzuca tego zachowania, dlatego też tylko od aplikacji zależy, czy będą używane wyłącznie poprawne znaki drukowalne.

Starsze systemy uniksowe ograniczały nazwy plików do 14 znaków. Obecnie wszystkie nowoczesne uniksowe systemy plików przeznaczają co najmniej 255 bajtów dla każdej nazwy pliku[2]. Wiele systemów plików dla Linuksa pozwala na użycie nawet dłuższych nazw[3].

Każdy katalog zawiera dwa specjalne katalogi: . (katalog „kropka") oraz .. (katalog „dwie kropki"). Katalog „kropka" to odwołanie do katalogu aktualnego. Katalog „dwie kropki" jest odwołaniem do katalogu, który bezpośrednio zawiera w sobie katalog aktualny. Na przykład, katalog */home/kidd/gold/..* jest takim samym katalogiem jak */home/kidd*. W przypadku katalogu głównego katalog „kropka" i „dwie kropki" są po prostu odwołaniami do niego — oznacza to, że katalogi /, /. oraz /.. są sobie równe. Zatem nawet katalog główny jest podkatalogiem — w tym przypadku, swoim własnym podkatalogiem.

[2] Należy zwrócić uwagę, że w ograniczeniu tym występuje *255 bajtów*, a nie *255 znaków*. Oczywiste jest, że znaki wielobajtowe zajmują więcej niż jeden z tych 255 bajtów.

[3] Oczywiście starsze systemy plików, takie jak FAT, które wspierane są przez system Linux, aby zapewnić wsteczną kompatybilność, wciąż posiadają swoje ograniczenia. W przypadku systemu plików FAT, ograniczenie to wynosi osiem znaków, po których następuje znak kropki, a następnie jeszcze trzy znaki. Definicja kropki jako znaku specjalnego wewnątrz systemu plików nie jest właściwa.

Aktualny katalog roboczy

Każdy proces posiada aktualny katalog, który odziedziczony zostaje na początku od procesu rodzicielskiego. Ten katalog znany jest pod nazwą *aktualnego katalogu roboczego* dla procesu (ang. *current working directory*, w skrócie *cwd*). Aktualny katalog roboczy jest punktem początkowym, od którego przez jądro systemu wykonywana jest analiza ścieżek względnych. Na przykład, jeśli aktualny katalog roboczy dla procesu równy jest */home/blackbeard*, a proces ten próbuje otworzyć plik o nazwie *parrot.jpg*, wówczas jądro w rzeczywistości przystąpi do otwarcia pliku */home/blackbeard/parrot.jpg*. I odwrotnie, jeśli proces będzie chciał otworzyć plik */usr/bin/mast*, wówczas jądro faktycznie otworzy */usr/bin/mast* — aktualny katalog roboczy nie ma wpływu na ścieżki bezwzględne (czyli na ścieżki rozpoczynające się od ukośnika).

Proces może odczytać i zmienić swój aktualny katalog roboczy.

Odczytywanie aktualnego katalogu roboczego

Zalecaną metodą dla odczytania aktualnego katalogu roboczego jest użycie funkcji systemowej `getcwd()`, która została zdefiniowana w standardzie POSIX:

```
#include <unistd.h>

char * getcwd (char *buf, size_t size);
```

Poprawne wywołanie funkcji `getcwd()` kopiuje aktualny katalog roboczy, reprezentowany w postaci ścieżki absolutnej, do bufora, wskazywanego przez parametr `buf`, którego długość określona jest w parametrze `size` i wyrażona w bajtach. Funkcja zwraca wskaźnik do bufora `buf`. W przypadku błędu funkcja zwraca `NULL` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EFAULT`
: Parametr `buf` jest nieprawidłowym wskaźnikiem.

`EINVAL`
: Parametr `size` wynosi `0`, lecz `buf` jest różny od `NULL`.

`ENOENT`
: Aktualny katalog roboczy nie jest już prawidłowy. Błąd ten może się pojawić, jeśli aktualny katalog zostanie usunięty.

`ERANGE`
: Parametr `size` jest zbyt mały, aby przechować aktualny katalog roboczy w buforze `buf`. Aplikacja musi przydzielić pamięć na większy bufor i ponownie wywołać funkcję.

Oto przykład użycia funkcji `getcwd()`:

```
char cwd[BUF_LEN];

if (!getcwd (cwd, BUF_LEN))
{
   perror ("getcwd");
   exit (EXIT_FAILURE);
}

printf ("Aktualny katalog roboczy = %s\n", cwd);
```

Standard POSIX ustala, że sposób działania funkcji `getcwd()` jest niezdefiniowany, jeśli `buf` równy jest `NULL`. W tym przypadku, biblioteka języka C dla Linuksa przydzieli miejsce w pamięci

na bufor o rozmiarze przekazanym w parametrze `size` i wyrażonym w bajtach, a następnie zapisze tam nazwę aktualnego katalogu roboczego. Jeśli rozmiar `size` jest równy zeru, biblioteka języka C przydzieli miejsce w pamięci na odpowiednio duży bufor, aby można było zapisać tam nazwę aktualnego katalogu roboczego. Gdy bufor przestanie być potrzebny, aplikacja powinna zwolnić za pomocą funkcji `free()` pamięć wcześniej zarezerwowaną dla niego. Zachowanie to jest specyficzne dla Linuksa, dlatego też aplikacje, które wymagają przenośności lub zgodności ze standardem POSIX, nie powinny opierać się na tej funkcjonalności. Ta możliwość pozwala jednak na bardzo proste użycie! Oto przykład:

```
char *cwd;

cwd = getcwd (NULL, 0);
if (!cwd)
{
   perror ("getcwd");
   exit (EXIT_FAILURE);
}

printf ("Aktualny katalog roboczy = %s\n", cwd);

free (cwd);
```

Biblioteka języka C dla Linuksa dostarcza także funkcji `get_current_dir_name()`, która działa w taki sam sposób jak funkcja `getcwd()`, dla której parametr `buf` równy jest `NULL`, a `size` zeru:

```
#define _GNU_SOURCE
#include <unistd.h>

char * get_current_dir_name (void);
```

Dlatego też poniższy fragment kodu działa tak samo jak poprzedni:

```
char *cwd;

cwd = get_current_dir_name ( );
if (!cwd)
{
   perror ("get_current_dir_name");
   exit (EXIT_FAILURE);
}

printf ("Aktualny katalog roboczy = %s\n", cwd);

free (cwd);
```

W starszych systemach BSD używano funkcji `getwd()`, która udostępniana jest w Linuksie ze względu na wsteczną kompatybilność:

```
#define _XOPEN_SOURCE_EXTENDED /* lub _BSD_SOURCE */
#include <unistd.h>

char * getwd (char *buf);
```

Wywołanie funkcji `getwd()` zapisuje nazwę aktualnego katalogu roboczego do bufora `buf`, który musi mieć długość nie mniejszą od stałej `PATH_MAX`, wyrażoną w bajtach. Funkcja zwraca wskaźnik do `buf` w przypadku sukcesu lub `NULL` w przypadku błędu. Na przykład:

```
char cwd[PATH_MAX];

if (!getwd (cwd))
{
   perror ("getwd");
```

```
    exit (EXIT_FAILURE);
}

printf ("Aktualny katalog roboczy = %s\n", cwd);
```

Z powodu konieczności zapewnienia przenośności i bezpieczeństwa, w aplikacjach nie powinno używać się funkcji `getwd()`; preferowane jest użycie `getcwd()`.

Zmiana aktualnego katalogu roboczego

Gdy użytkownik loguje się po raz pierwszy do systemu, proces logowania ustawia jego aktualny katalog roboczy na katalog własny, zgodnie z wpisem znajdującym się w pliku */etc/passwd*. Czasem jednak jest wymagane, aby zmienić aktualny katalog roboczy procesu. Na przykład, powłoka systemowa może wykonać tę czynność, gdy użytkownik uruchomi polecenie *cd*.

Linux dostarcza dwóch funkcji systemowych w celu zmiany aktualnego katalogu roboczego — jedną, która akceptuje ścieżkę do katalogu i drugą, używającą deskryptora pliku reprezentującego otwarty katalog:

```
#include <unistd.h>

int chdir (const char *path);
int fchdir (int fd);
```

Wywołanie funkcji `chdir()` zmienia aktualny katalog roboczy na ścieżkę podaną w parametrze `path`, która może być bezwzględna lub względna. Podobnie, wywołanie funkcji `fchdir()` zmienia aktualny katalog roboczy na ścieżkę reprezentowaną przez deskryptor pliku `fd`, który musi być otwarty i wskazywać na katalog docelowy. W przypadku sukcesu obie funkcje zwracają wartość zero; w przypadku błędu zwracają `-1`, a funkcja `chdir()` dodatkowo ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnienia szukania dla jednego z elementów katalogu przekazanego w parametrze `path`.

`EFAULT`
: Parametr `path` jest niepoprawnym wskaźnikiem.

`EIO`
: Wystąpił wewnętrzny błąd operacji wejścia i wyjścia.

`ELOOP`
: Jądro napotkało zbyt wiele dowiązań symbolicznych podczas analizowania ścieżki `path`.

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa.

`ENOENT`
: Katalog, wskazywany przez ścieżkę `path`, nie istnieje.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOTDIR`
: Jeden lub więcej elementów ścieżki `path` nie jest katalogiem.

Funkcja `fchdir( )` ustawia zmienną `errno` na jedną z poniższych wartości:

EACCESS
: Proces wywołujący nie posiada uprawnienia szukania dla katalogu reprezentowanego przez parametr `path` (na przykład nie został ustawiony bit wykonania). Ten błąd ma miejsce wówczas, gdy najwyższy katalog posiada uprawnienia do odczytu, lecz nie ma uprawnień do wykonania; funkcja `open()` wykonuje się prawidłowo, lecz `fchdir()` już nie.

EBADF
: Parametr `fd` nie jest otwartym deskryptorem pliku.

W zależności od systemu plików funkcje mogą zwracać dodatkowe kody błędów.

Powyższe funkcje systemowe działają tylko dla aktualnego procesu. W systemie Unix nie istnieje mechanizm, który pozwalałby na zmianę aktualnego katalogu roboczego dla innego procesu. Dlatego też polecenie *cd*, które istnieje w powłoce systemowej, nie może być oddzielnym procesem (jak większość innych poleceń) wywołującym funkcję `chdir()` z pierwszym parametrem linii poleceń, a następnie kończącym swoje działanie. Polecenie *cd* musi być specjalnym poleceniem, wbudowanym w powłokę systemową. Jego wykonanie powoduje, że powłoka sama wywołuje funkcję `chdir()`, zmieniając własny, aktualny katalog roboczy.

Najbardziej powszechny sposób użycia funkcji `getcwd()` polega na zapamiętaniu aktualnego katalogu roboczego, aby proces mógł wrócić do niego w przyszłości. Na przykład:

```
char *swd;
int ret;

/* zapamiętaj aktualny katalog roboczy */
swd = getcwd (NULL, 0);
if (!swd)
{
   perror ("getcwd");
   exit (EXIT_FAILURE);
}

/* przejdź do innego katalogu */
ret = chdir (some_other_dir);
if (ret)
{
   perror ("chdir");
   exit (EXIT_FAILURE);
}

/* wykonaj jakieś czynności w nowym katalogu... */

/* wróć do zapamiętanego katalogu */
ret = chdir (swd);
if (ret)
{
   perror ("chdir");
   exit (EXIT_FAILURE);
}

free (swd);
```

Jednak lepszym rozwiązaniem jest użycie funkcji `open()`, by dostać się do aktualnego katalogu, a następnie wykonanie funkcji `fchdir()`, by później do niego wrócić. Ta operacja jest szybsza, ponieważ jądro nie przechowuje w pamięci ścieżki do aktualnego katalogu roboczego, lecz wyłącznie i-węzeł. Zgodnie z tym, gdy tylko użytkownik wywoła funkcję `getcwd()`, jądro musi utworzyć ścieżkę, przechodząc po strukturze katalogu. I odwrotnie, otwieranie aktualnego katalogu roboczego jest bardziej opłacalne, ponieważ jądro posiada już dostępny jego i-węzeł,

dlatego też nazwa ścieżki, zawierająca tekst możliwy do odczytania przez człowieka, nie jest wymagana, aby otworzyć plik. Poniższy fragment kodu używa opisanej przed chwilą metody:

```
int swd_fd;
swd_fd = open (".", O_RDONLY);
if (swd_fd == -1)
{
   perror ("open");
   exit (EXIT_FAILURE);
}

/* przejdź do innego katalogu */
ret = chdir (some_other_dir);
if (ret)
{
   perror ("chdir");
   exit (EXIT_FAILURE);
}

/* wykonaj jakieś czynności w nowym katalogu... */

/* wróć do zapamiętanego katalogu */
ret = fchdir (swd_fd);
if (ret)
{
   perror ("fchdir");
   exit (EXIT_FAILURE);
}

/* zamknij deskryptor pliku dla katalogu */
ret = close (swd_fd);
if (ret)
{
   perror ("close");
   exit (EXIT_FAILURE);
}
```

W ten sposób w powłoce systemowej zaimplementowane jest zapamiętywanie poprzedniego katalogu (na przykład dla powłoki *bash*, w przypadku wykonywania polecenia *cd -*).

Proces, który nie potrzebuje do swojego działania aktualnego katalogu roboczego, takiego jak demon, zazwyczaj ustawia jego wartość na katalog główny za pomocą funkcji `chdir("/")`. Aplikacja, która kontaktuje się z użytkownikiem i jego danymi, taka jak edytor tekstu, przeważnie ustawia swój aktualny katalog roboczy na katalog własny użytkownika lub na specjalny katalog z dokumentami. Ponieważ aktualne katalogi robocze mają znaczenie tylko w powiązaniu ze ścieżkami względnymi, są używane przede wszystkim przez programy użytkowe, uruchamiane przez użytkownika z linii poleceń powłoki systemowej.

Tworzenie katalogów

Linux udostępnia pojedynczą funkcję systemową dla tworzenia nowych katalogów, która zdefiniowana jest w standardzie POSIX:

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

int mkdir (const char *path, mode_t mode);
```

Poprawne wywołanie funkcji `mkdir()` tworzy katalog `path`, który może znajdować się w ścieżce względnej lub bezwzględnej. Katalog ten uzyskuje uprawnienia określone w parametrze `mode` (zmodyfikowane przez aktualną wartość maski uprawnień dla nowo tworzonych plików — `umask`). W przypadku sukcesu, funkcja zwraca wartość zero.

Parametr `mode` modyfikowany jest w sposób standardowy, za pomocą aktualnej wartości maski uprawnień dla nowo tworzonych plików, uwzględniając również specjalne bity uprawnień, specyficzne dla systemu operacyjnego: bity uprawnień nowo utworzonego katalogu w Linuksie są równe wartości `(mode & ~umask & 01777)`. Inaczej mówiąc, maska uprawnień dla procesu narzuca ograniczenia, które nie mogą zostać zmienione podczas wywołania funkcji `mkdir()`. Jeśli katalog nadrzędny nowo utworzonego katalogu posiada ustawiony bit SGID lub jeśli system plików został zamontowany przy użyciu semantyki grupy dla systemu operacyjnego BSD, wówczas nowy katalog odziedziczy przynależność do grup od swojego katalogu nadrzędnego. W przeciwnym razie dla nowego katalogu zostanie użyty efektywny identyfikator grupy procesu.

W przypadku błędu funkcja `mkdir()` zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Katalog nadrzędny nie ma uprawnień do zapisu dla aktualnego procesu lub jeden albo więcej elementów ścieżki `path` nie mogą zostać przeszukane.

`EEXIST`
: Ścieżka `path` już istnieje (i niekoniecznie jest katalogiem).

`EFAULT`
: Parametr `path` jest błędnym wskaźnikiem.

`ELOOP`
: Jądro napotkało zbyt wiele dowiązań symbolicznych podczas analizowania ścieżki `path`.

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa.

`ENOENT`
: Element w ścieżce `path` nie istnieje lub jest zawieszonym dowiązaniem symbolicznym.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci dla jądra, aby zakończyć tę operację.

`ENOSPC`
: W urządzeniu, zawierającym ścieżkę `path`, zabrakło wolnego obszaru lub przekroczony został limit miejsca na dysku dla użytkownika.

`ENOTDIR`
: Jeden lub więcej elementów ścieżki `path` nie jest katalogiem.

`EPERM`
: System plików, zawierający ścieżkę `path`, nie wspiera operacji tworzenia katalogów.

`EROFS`
: System plików, zawierający ścieżkę `path`, został zamontowany w trybie tylko do odczytu.

Usuwanie katalogów

W standardzie POSIX zdefiniowana jest funkcja o przeciwnym działaniu do `mkdir()`, posiadająca nazwę `rmdir()` i umożliwiająca usuwanie katalogu z hierarchii systemu plików:

```
#include <unistd.h>

int rmdir (const char *path);
```

W przypadku sukcesu, funkcja `rmdir()` usuwa katalog `path` z systemu plików i zwraca `0`. Katalog, określony w parametrze `path`, nie może zawierać żadnych elementów, za wyjątkiem odnośników do katalogów bieżącego i nadrzędnego. Nie istnieje żadna funkcja systemowa, która posiadałaby zaimplementowaną operację rekurencyjnego usuwania katalogów, działająca tak, jak funkcjonuje polecenie *rm -r*. Takie narzędzie musi ręcznie przeprowadzić operację przeglądania systemu plików, poczynając od usunięcia wszystkich plików i katalogów z najniższego poziomu, a następnie przesuwając się w kierunku katalogu głównego; funkcja `rmdir()` może zostać użyta na każdym z tych poziomów, aby usunąć dany katalog, gdy tylko pliki, znajdujące się w nim, zostaną skasowane.

W przypadku błędu funkcja `rmdir()` zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Nie istnieje uprawnienie do zapisu do katalogu nadrzędnego dla ścieżki `path` lub też jeden albo więcej katalogów, tworzących ścieżkę `path`, nie mogą zostać przeszukane.

`EEXIST`
: Ścieżka `path` jest aktualnie używana przez system i nie może zostać usunięta. W systemie Linux taki problem może pojawić się tylko wówczas, gdy `path` jest punktem montowania lub katalogiem głównym (katalogi główne nie muszą być punktami montowania dzięki funkcji `chroot()`!).

`EFAULT`
: Parametr `path` jest błędnym wskaźnikiem.

`EINVAL`
: Końcowym elementem ścieżki `path` jest odnośnik do katalogu bieżącego (katalog „kropka”).

`ELOOP`
: Jądro napotkało zbyt wiele dowiązań symbolicznych podczas analizowania ścieżki `path`.

`ENAMETOOLONG`
: Ścieżka `path` jest zbyt długa.

`ENOENT`
: Element w ścieżce `path` nie istnieje lub jest zawieszonym dowiązaniem symbolicznym.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci dla jądra, aby zakończyć tę operację.

`ENOTDIR`
: Jeden lub więcej elementów ścieżki `path` nie jest katalogiem.

`ENOTEMPTY`
: Katalog `path` zawiera również inne elementy, a nie tylko specjalne odwołania do katalogów bieżącego i nadrzędnego.

EPERM
: Katalog nadrzędny dla ścieżki `path` posiada ustawiony bit sticky (`S_ISVTX`), lecz efektywny identyfikator użytkownika dla procesu nie jest identyfikatorem użytkownika dla wspomnianego katalogu ani dla samej ścieżki `path`, a proces nie posiada uprawnienia `CAP_FOWNER`. Innym powodem tego błędu może być to, że system plików, zawierający ścieżkę `path`, nie zezwala na usuwanie katalogów.

EROFS
: System plików, zawierający ścieżkę `path`, został zamontowany w trybie tylko do odczytu.

Użycie funkcji jest proste:

```
int ret;

/* usunięcie katalogu /home/barbary/maps */
ret = rmdir ("/home/barbary/maps");
if (ret)
   perror ("rmdir");
```

Odczytywanie zawartości katalogu

POSIX definiuje zestaw funkcji służących do odczytywania zawartości katalogów — to znaczy, uzyskiwania listy plików, które znajdują się w danym katalogu. Funkcje te są użyteczne w przypadku implementowania polecenia *ls* lub okna dialogu z graficznym wskaźnikiem postępu operacji zapisu plików, a także w przypadku, gdy należy przetworzyć każdy plik z danego katalogu lub przeszukać pliki w poszukiwaniu określonej nazwy.

Aby rozpocząć czytanie zawartości katalogu, należy stworzyć *strumień katalogu* (ang. *directory stream*), który reprezentowany jest przez obiekt `DIR`:

```
#include <sys/types.h>
#include <dirent.h>

DIR * opendir (const char *name);
```

Poprawne wywołanie funkcji `opendir()` tworzy strumień katalogu, reprezentujący katalog, którego nazwa podana jest w parametrze `name`.

Strumień katalogu jest w przybliżeniu deskryptorem pliku reprezentującym otwarty katalog, pewne metadane oraz bufor przechowujący zawartość katalogu. Możliwe jest odczytanie deskryptora pliku, odpowiadającego danemu strumieniowi katalogu:

```
#define _BSD_SOURCE /* lub _SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>

int dirfd (DIR *dir);
```

Poprawne wywołanie funkcji `dirfd()` zwraca deskryptor pliku, który reprezentuje strumień katalogu `dir`. W przypadku błędu, funkcja zwraca `-1`. Ponieważ funkcje dla strumienia katalogu używają wewnętrznie tego deskryptora pliku, w programach należy używać jedynie takich wywołań, które nie modyfikują pozycji w pliku. Funkcja `dirfd()` jest rozszerzeniem systemu BSD i nie została zawarta w standardzie POSIX; programiści, którzy chcą informować o zgodności swoich programów ze standardem POSIX, powinni jej unikać.

Czytanie ze strumienia katalogu

Gdy strumień katalogu został już utworzony za pomocą funkcji `opendir()`, program może rozpocząć odczytywanie wpisów z katalogu. Aby to zrealizować, należy użyć funkcji `readdir()`, która zwraca po jednym elemencie katalogu z danego obiektu `DIR`:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent * readdir (DIR *dir);
```

Poprawne wywołanie funkcji `readdir()` zwraca kolejny wpis z katalogu reprezentowanego przez parametr `dir`. Struktura `dirent` określa element katalogu. Jest ona zdefiniowana w pliku nagłówkowym `<dirent.h>` i w przypadku systemu Linux wygląda następująco:

```
struct dirent
{
   ino_t d_ino; /* numer i-węzła */
   off_t d_off; /* odległość od następnego wpisu */
   unsigned short d_reclen; /* długość rekordu */
   unsigned char d_type; /* typ pliku */
   char d_name[256]; /* nazwa pliku */
};
```

POSIX wymaga istnienia tylko pola `d_name`, które określa nazwę pojedynczego pliku wewnątrz katalogu. Inne pola są opcjonalne lub specyficzne dla Linuksa. W aplikacjach, które wymagają spełnienia warunków przenośności do innych systemów lub zgodności ze standardem POSIX, należy używać wyłącznie pola `d_name`.

Aplikacje wywołują wielokrotnie funkcję `readdir()`, uzyskując kolejne pliki z katalogu, dopóki nie znajdą pliku, którego szukają lub póki cały katalog nie zostanie odczytany, co sygnalizowane jest zwróceniem wartości `NULL` przez tę funkcję.

W przypadku błędu, funkcja `readdir()` również zwraca `NULL`. Aby odróżnić sytuację błędu od odczytania wszystkich plików z katalogu, aplikacje muszą ustawiać zmienną `errno` na wartość `0` przed każdym wywołaniem funkcji `readdir()`, a następnie sprawdzać zarówno kod powrotu, jak i wartość zmiennej `errno`. Jedyną wartością, ustawianą w zmiennej `errno` przez funkcję `readdir()`, jest kod `EBADF`, który oznacza, że strumień katalogu `dir` jest nieprawidłowy. Dlatego też w przypadku wielu aplikacji nie sprawdza się, czy wywołanie funkcji zakończyło się błędem, natomiast zakłada, że zwrócona wartość `NULL` oznacza, że nie ma więcej plików do odczytania.

Zamykanie strumienia katalogu

Aby zamknąć strumień katalogu, otwarty za pomocą `opendir()`, należy użyć funkcji `closedir()`:

```
#include <sys/types.h>
#include <dirent.h>

int closedir (DIR *dir);
```

Poprawne wywołanie funkcji `closedir()` zamyka strumień katalogu, reprezentowany przez parametr `dir`, a następnie zwraca `0`. Zamknięty zostaje również odpowiedni deskryptor pliku dla strumienia katalogu. W przypadku błędu, funkcja zwraca `−1` oraz ustawia zmienną `errno` na jedyną możliwą wartość kodu równą `EBADF`, która oznacza, że `dir` nie jest otwartym strumieniem katalogu.

Poniższy fragment kodu definiuje funkcję `find_file_in_dir()`, która używa `readdir()`, aby odnaleźć dany plik w określonym katalogu. Jeśli plik znajduje się w tym katalogu, funkcja zwraca `0`. W przeciwnym razie zwraca wartość niezerową:

```
/*
* find_file_in_dir - w katalogu 'path' szuka pliku o nazwie 'file'.
*
* Zwraca 0, jeśli plik 'file' znajduje się w katalogu 'path',
* natomiast w przeciwnym przypadku zwraca wartość niezerową.
*/
int find_file_in_dir (const char *path, const char *file)
{
   struct dirent *entry;
   int ret = 1;
   DIR *dir;

   dir = opendir (path);

   errno = 0;
   while ((entry = readdir (dir)) != NULL)
   {
      if (!strcmp(entry->d_name, file))
      {
         ret = 0;
         break;
      }
   }

   if (errno && !entry)
      perror ("readdir");

   closedir (dir);
   return ret;
}
```

Funkcje systemowe służące do odczytywania zawartości katalogu

Poprzednio omówione funkcje służące do odczytywania zawartości katalogu są zdefiniowane w standardzie POSIX i udostępnione w bibliotece języka C. Wewnętrznie używają one dwóch funkcji systemowych: `readdir()` i `getdents()`, które zostaną poniżej przedstawione w celu zapewnienia kompletności informacji:

```
#include <unistd.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/unistd.h>
#include <errno.h>

/*
* Funkcji tych nie zdefiniowano dla przestrzeni użytkownika:
* aby uzyskać do nich dostęp, należy użyć makra _syscall3().
*/
int readdir (unsigned int fd, struct dirent *dirp, unsigned int count);
int getdents (unsigned int fd, struct dirent *dirp, unsigned int count);
```

Programista nie powinien nawet pomyśleć o używaniu tych funkcji! Są prymitywne i nieprzenośne. Zamiast nich, w aplikacjach z przestrzeni użytkownika należy stosować funkcje z biblioteki języka C o nazwach `opendir()`, `readdir()` i `closedir()`.

Dowiązania

Wcześniej, w trakcie omawiania katalogów, stwierdzono, że każde odwzorowanie nazwy na i-węzeł nazywane jest *dowiązaniem* (ang. *link*). Posiadając taką prostą definicję, iż dowiązanie jest po prostu nazwą na liście (lub w katalogu), która wskazuje na i-węzeł, wydawałoby się, że nie ma przeszkody, aby nie istniało wiele dowiązań do tego samego i-węzła. Oznacza to, że do pojedynczego i-węzła (czyli pojedynczego pliku) mogłyby istnieć odsyłacze na przykład z miejsc */etc/customs* i */var/run/ledger*.

I tak faktycznie jest, z jednym zastrzeżeniem. Ponieważ dowiązania odwzorowane są na i-węzły, a i-węzły są specyficzne dla danego systemu plików, dlatego też */etc/customs* oraz */var/run/ledger* muszą znajdować się w tym samym systemie plików. Wewnątrz pojedynczego systemu plików może istnieć bardzo wiele dowiązań do dowolnego pliku. Jedynym ograniczeniem jest rozmiar typu danych całkowitych, używany w celu zapamiętania liczby dowiązań. Wśród różnych dowiązań nie istnieje żadne, które byłoby „pierwotnym" lub „głównym". Wszystkie posiadają identyczny status, wskazując na ten sam plik.

Tego typu dowiązania zwane są *dowiązaniami twardymi* (ang. *hard links*). Pliki mogą nie posiadać żadnego dowiązania twardego, ale również mogą mieć jedno dowiązanie lub więcej. Większość plików posiada licznik dowiązań równy 1 (oznacza to, że są wskazywane przez pojedynczy wpis w katalogu), lecz niektóre mają dwa dowiązania lub więcej. Pliki z licznikiem dowiązań równym zeru nie posiadają odpowiednich wpisów w katalogu w danym systemie plików. Gdy licznik dowiązań dla pliku osiąga wartość zero, plik zostaje zwolniony, a zajmowane wcześniej przez niego bloki na dysku zostają udostępnione do ponownego użycia[4]. Taki plik pozostaje jednak w systemie, jeśli jest on otwarty przez jakiś proces, który go używa. Gdy tylko przestają istnieć procesy używające otwartego pliku, zostaje on usunięty.

Jądro Linuksa implementuje to zachowanie poprzez użycie licznika dowiązań oraz licznika użycia. *Licznik użycia* (ang. *usage count*) jest listą z zapisaną liczbą procesów, w których plik jest otwarty. Plik nie zostanie usunięty z systemu plików, dopóki liczniki dowiązań i użycia nie będą równe zeru.

Innym rodzajem dowiązania jest *dowiązanie symboliczne* (ang. *symbolic link*). Nie jest to odwzorowanie w systemie plików, lecz wskazanie wyższego poziomu, które jest interpretowane podczas pracy systemu. Takie dowiązania mogą obejmować różne systemy plików — zostaną one wkrótce omówione.

Dowiązania twarde

Funkcja systemowa `link()`, będąca jedną z pierwotnych funkcji systemu Unix, zdefiniowana jest obecnie w standardzie POSIX. Funkcja ta tworzy nowe dowiązanie dla istniejącego pliku:

[4] Szukanie plików, których licznik dowiązań jest równy zeru, posiadających bloki oznaczone jako przydzielone, jest podstawowym zadaniem programu *fsck*, który sprawdza stan systemu plików. Taka sytuacja może się zdarzyć, gdy plik jest usunięty, natomiast pozostaje jeszcze w stanie otwartym, a zanim zostanie zamknięty, system operacyjny ulegnie załamaniu. Jądro nie jest w stanie oznaczyć tych bloków w systemie plików jako zwolnionych, dlatego też pojawia się rozbieżność. Powyższy rodzaj błędu zostaje wyeliminowany po zastosowaniu systemów plików z dziennikiem.

```
#include <unistd.h>

int link (const char *oldpath, const char *newpath);
```

Poprawne wywołanie funkcji `link()` tworzy nowe dowiązanie o ścieżce `newpath` dla istniejącego pliku, określonego w parametrze `oldpath`, a następnie zwraca `0`. Po zakończeniu działania funkcji obie ścieżki `oldpath` oraz `newpath` wskazują na ten sam plik — w rzeczywistości nie można nawet ustalić, która z nich była „pierwotna".

W przypadku błędu funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnień wyszukiwania w jednym z elementów ścieżki `oldpath` lub nie posiada uprawnień zapisu do katalogu zawierającego ścieżkę `newpath`.

`EEXIST`
: Ścieżka `newpath` już istnieje — funkcja `link()` nie nadpisze żadnego istniejącego elementu katalogu.

`EFAULT`
: Parametry `oldpath` lub `newpath` są niepoprawnymi wskaźnikami.

`EIO`
: Wystąpił wewnętrzny błąd operacji wejścia i wyjścia (jest to poważny problem!).

`ELOOP`
: Napotkano zbyt wiele dowiązań symbolicznych podczas analizy ścieżek `oldpath` lub `newpath`.

`EMLINK`
: Dla i-węzła, reprezentowanego przez `oldpath`, osiągnięto już maksymalną liczbę dowiązań, które na niego wskazują.

`ENAMETOOLONG`
: Nazwy `oldpath` lub `newpath` są zbyt długie.

`ENOENT`
: Element ścieżki `oldpath` lub `newpath` nie istnieje.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOSPC`
: W urządzeniu, zawierającym ścieżkę `newpath`, nie ma już wolnego miejsca na nowy element katalogu.

`ENOTDIR`
: Element ścieżek `oldpath` lub `newpath` nie jest katalogiem.

`EPERM`
: System plików, zawierający ścieżkę `newpath`, nie zezwala na tworzenie nowych dowiązań twardych lub ścieżka `oldpath` jest katalogiem.

`EROFS`
: Ścieżka `newpath` znajduje się w systemie plików z uprawnieniami tylko do odczytu.

`EXDEV`
: Ścieżki `newpath` i `oldpath` nie znajdują się w tym samym zamontowanym systemie plików (Linux zezwala, aby pojedynczy system plików był zamontowany w wielu miejscach, lecz nawet w tym przypadku dowiązania twarde nie mogą się odwoływać poprzez różne punkty montowania).

Poniższy przykład tworzy nowy element katalogu o nazwie *pirate*, następnie odwzorowuje go do tego samego i-węzła (stąd także do tego samego pliku), na który wskazuje istniejący już plik *privateer*. Oba elementy znajdują się w katalogu */home/kidd*:

```
int ret;

/*
* tworzy nowy element katalogu o nazwie '/home/kidd/privateer',
* wskazujący na ten sam i-węzeł co plik '/home/kidd/pirate'
*/
ret = link ("/home/kidd/privateer", /home/kidd/pirate");
if (ret)
  perror ("link");
```

Dowiązania symboliczne

Dowiązania symboliczne (ang. *symbolic links*, w skrócie *symlinks*), zwane także *dowiązaniami miękkimi* (ang. *soft links*), są podobne do dowiązań twardych, gdyż zarówno jedne, jak i drugie wskazują na pliki w systemie plików. Dowiązanie symboliczne różni się jednak od dowiązania twardego tym, że nie jest wyłącznie dodatkowym wpisem w katalogu, lecz zupełnie odmiennym rodzajem pliku. Ten plik specjalny zawiera ścieżkę do *innego* pliku, zwanego *plikiem docelowym* dowiązania symbolicznego. Podczas działania systemu jądro na bieżąco zastępuje ścieżkę do dowiązania symbolicznego ścieżką do pliku docelowego (chyba że zostaną użyte różne wersje `l` funkcji systemowych, takie jak `lstat()`, przetwarzające samo dowiązanie symboliczne, a nie plik docelowy). Tak więc dwa dowiązania twarde, wskazujące na ten sam plik, są nieodróżnialne od siebie, można natomiast łatwo pokazać różnicę między dowiązaniem symbolicznym, a jego plikiem docelowym.

Dowiązanie symboliczne może być względne lub bezwzględne. Może również zawierać omówiony już wcześniej specjalny katalog „kropka", określający bieżące miejsce, w którym znajduje się dane dowiązanie, a także katalog „dwie kropki", oznaczający katalog bezpośrednio nadrzędny względem aktualnego.

Dowiązania miękkie, w przeciwieństwie do twardych, mogą obejmować różne systemy plików. W rzeczywistości mogą wskazywać na cokolwiek, a dowiązania symboliczne na pliki istniejące (jest to najczęstszy przypadek) lub nieistniejące. Ten drugi rodzaj dowiązania zwany jest *zawieszonym dowiązaniem symbolicznym* (ang. *dangling symlink*). Czasem takie dowiązania są niepożądane (na przykład, gdy plik docelowy został usunięty, natomiast samo dowiązanie pozostało), lecz istnieją w określonym celu.

Funkcja systemowa, tworząca dowiązanie symboliczne, jest bardzo podobna do bliźniaczej funkcji obsługującej dowiązanie twarde:

```
#include <unistd.h>

int symlink (const char *oldpath, const char *newpath);
```

Poprawne wywołanie funkcji `symlink()` tworzy dowiązanie symboliczne o ścieżce `newpath`, wskazujące na plik docelowy `oldpath`, a następnie zwraca `0`.

W przypadku błędu funkcja `symlink()` zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnień wyszukiwania w jednym z elementów ścieżki `oldpath` lub nie posiada uprawnień zapisu do katalogu zawierającego ścieżkę `newpath`.

`EEXIST`
: Ścieżka `newpath` już istnieje — funkcja `symlink()` nie nadpisze istniejącego elementu katalogu.

`EFAULT`
: Parametry `oldpath` lub `newpath` są niepoprawnymi wskaźnikami.

`EIO`
: Wystąpił wewnętrzny błąd operacji wejścia i wyjścia (jest to poważny problem!).

`ELOOP`
: Napotkano zbyt wiele dowiązań symbolicznych podczas analizy ścieżek `oldpath` lub `newpath`.

`EMLINK`
: Dla i-węzła, reprezentowanego przez `oldpath`, osiągnięto już maksymalną liczbę dowiązań, które na niego wskazują.

`ENAMETOOLONG`
: Nazwy `oldpath` lub `newpath` są zbyt długie.

`ENOENT`
: Element ścieżki `oldpath` lub `newpath` nie istnieje.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOSPC`
: W urządzeniu, zawierającym ścieżkę `newpath`, nie ma wolnego miejsca na nowy element katalogu.

`ENOTDIR`
: Element ścieżek `oldpath` lub `newpath` nie jest katalogiem.

`EPERM`
: System plików, zawierający ścieżkę `newpath`, nie zezwala na tworzenie nowych dowiązań symbolicznych.

`EROFS`
: Ścieżka `newpath` znajduje się w systemie plików z uprawnieniami tylko do odczytu.

Poniższy fragment kodu wygląda prawie identycznie jak poprzednio przedstawiony przykład, lecz tworzy nowy element */home/kidd/pirate* w postaci dowiązania symbolicznego (w przeciwieństwie do dowiązania twardego) do pliku */home/kidd/privateer*:

```
int ret;

/*
* tworzy dowiązanie symboliczne o nazwie '/home/kidd/privateer',
* które wskazuje na '/home/kidd/pirate'
```

```
*/
ret = symlink ("/home/kidd/privateer", "/home/kidd/pirate");
if (ret)
   perror ("symlink");
```

Usuwanie elementów z systemu plików

Przeciwieństwem tworzenia jest usuwanie elementów, polegające na kasowaniu ścieżek z systemu plików. Operacja ta wykonywana jest przy użyciu pojedynczej funkcji systemowej `unlink()`:

```
#include <unistd.h>

int unlink (const char *pathname);
```

Poprawne wywołanie funkcji `unlink()` usuwa element o nazwie `pathname` z systemu plików i zwraca `0`. Jeśli nazwa ta była ostatnim odwołaniem do pliku, zostaje on usunięty z systemu plików. Jeśli jednak plik ten jest otwarty przez jakiś proces, jądro nie usunie go z systemu, dopóki nie zostanie on przez ten proces zamknięty. Jeśli plik nie jest już otwarty przez żaden proces, zostanie usunięty.

Jeśli `pathname` jest dowiązaniem symbolicznym, usunięte zostanie samo dowiązanie, a nie plik docelowy.

Jeśli `pathname` dotyczy innego rodzaju pliku specjalnego, takiego jak urządzenie, kolejka FIFO lub gniazdo, plik ten zostaje usunięty z systemu plików, lecz procesy, które go otwarły, mogą w dalszym ciągu z niego korzystać.

W przypadku błędu funkcja `unlink()` zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnień zapisu do katalogu nadrzędnego względem ścieżki `pathname` lub nie posiada uprawnień wyszukiwania w jednym z jej elementów.

`EFAULT`
: Parametr `pathname` jest niepoprawnym wskaźnikiem.

`EIO`
: Wystąpił wewnętrzny błąd operacji wejścia i wyjścia (jest to poważny problem!).

`ELOOP`
: Napotkano zbyt wiele dowiązań symbolicznych podczas analizy ścieżki `pathname`.

`ENAMETOOLONG`
: Nazwa `pathname` jest zbyt długa.

`ENOENT`
: Element ścieżki `pathname` nie istnieje.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOTDIR`
: Element ścieżki `pathname` nie jest katalogiem.

`EPERM`
: System plików nie zezwala na usuwanie plików.

EROFS
Ścieżka pathname znajduje się w systemie plików z uprawnieniami tylko do odczytu.

Funkcja unlink() nie usuwa katalogów. W tym celu w aplikacjach powinno używać się funkcji rmdir(), która została wcześniej omówiona (w rozdziale Usuwanie katalogów).

By ułatwić wykonanie operacji bezmyślnego usunięcia pliku o dowolnym typie, język C dostarcza funkcji remove():

```
#include <stdio.h>

int remove (const char *path);
```

Poprawne wywołanie funkcji remove() usuwa element path z systemu plików i zwraca zero. Jeśli element path jest plikiem, funkcja remove() wywołuje unlink(); jeśli path jest katalogiem, następuje wywołanie rmdir().

W przypadku błędu, funkcja remove() zwraca −1 oraz odpowiednio ustawia zmienną errno na wartość z zakresu kodów błędów zwracanych przez funkcje unlink() lub rmdir().

Kopiowanie i przenoszenie plików

Dwiema najbardziej podstawowymi operacjami, dotyczącymi plików, są kopiowanie i przenoszenie, zwykle realizowane za pomocą poleceń *cp* i *mv*. Na poziomie systemu plików *kopiowanie* jest czynnością powielania zawartości danego pliku, aby możliwe było odwołanie się do niej za pomocą innej ścieżki. Różni się to od tworzenia nowego dowiązania twardego, ponieważ zmiany w jednym pliku nie będą uwzględnione w drugim — oznacza to, że obecnie będą istnieć dwie oddzielne kopie pliku, posiadające (co najmniej) dwa różne wpisy w katalogu. *Przenoszenie* — przeciwnie — jest czynnością polegającą na zmianie nazwy w katalogu, która reprezentuje dany plik. Czynność ta nie powoduje utworzenia drugiej kopii pliku.

Kopiowanie

Chociaż jest to dla niektórych zaskakujące, Unix nie zawiera żadnej funkcji systemowej ani bibliotecznej, która pozwalałaby na kopiowanie plików i katalogów. Zamiast tego programy użytkowe, takie jak *cp* lub zarządca plików Nautilus działający w środowisku GNOME, wykonują te operacje samodzielnie.

Aby skopiować plik src do pliku dst, wymagane jest przeprowadzenie następujących działań:

1. Otworzyć plik src.
2. Otworzyć plik dst, tworząc go lub jeśli istnieje, obciąć do długości zero.
3. Odczytać pakiet danych z pliku src do pamięci.
4. Zapisać ten pakiet danych do pliku dst.
5. Kontynuować kroki 3. i 4. aż do momentu, gdy cały plik src zostanie skopiowany do dst.
6. Zamknąć plik dst.
7. Zamknąć plik src.

W przypadku kopiowania katalogu nowy katalog (a także jego podkatalogi) musi najpierw zostać utworzony za pomocą funkcji `mkdir()`; następnie każdy plik zostaje oddzielnie skopiowany.

Przenoszenie

W przeciwieństwie do kopiowania Unix rzeczywiście udostępnia funkcję systemową, pozwalającą na przenoszenie plików. Standard ANSI C pozwolił na stosowanie tej funkcji dla plików, natomiast POSIX dodał możliwość przenoszenia katalogów:

```
#include <stdio.h>

int rename (const char *oldpath, const char *newpath);
```

Poprawne wywołanie funkcji `rename()` zmienia nazwę `oldpath` na `newpath`. Zawartość pliku oraz i-węzeł pozostają niezmienione. Obie ścieżki `oldpath` i `newpath` muszą znajdować się w tym samym systemie plików[5]; jeśli nie będzie to spełnione, wywołanie funkcji nie powiedzie się. Programy użytkowe, takie jak *mv*, muszą obsługiwać ten przypadek poprzez wykonywanie kopiowania i usuwania elementów.

W przypadku sukcesu, funkcja `rename()` zwraca `0`, a plik, który przedtem posiadał nazwę `oldpath`, obecnie reprezentowany jest przez nazwę `newpath`. W przypadku błędu, funkcja zwraca `−1`, nie modyfikuje parametrów `oldpath` i `newpath` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Proces wywołujący nie posiada uprawnień zapisu do katalogu nadrzędnego względem ścieżki `oldpath` lub `newpath`, uprawnień wyszukiwania w jednym z elementów `oldpath` lub `newpath` bądź uprawnień zapisu dla `oldpath`, gdy ścieżka ta jest katalogiem.

`EBUSY`
: Ścieżka `oldpath` lub `newpath` jest punktem montowania.

`EFAULT`
: Parametry `oldpath` lub `newpath` są niepoprawnymi wskaźnikami.

`EINVAL`
: Ścieżka `newpath` jest zawarta w ścieżce `oldpath`, dlatego też zmiana nazwy jednej na drugą mogłaby spowodować, że `oldpath` stałaby się swoim własnym podkatalogiem.

`EISDIR`
: Ścieżka `newpath` już istnieje i jest katalogiem, natomiast `oldpath` nim nie jest.

`ELOOP`
: Napotkano zbyt wiele dowiązań symbolicznych podczas analizy ścieżek `oldpath` lub `newpath`.

`EMLINK`
: Dla ścieżki `oldpath` osiągnięto już maksymalną liczbę dowiązań, które na nią wskazują lub `oldpath` jest katalogiem, a ścieżka `newpath` posiada już maksymalną liczbę dowiązań.

`ENAMETOOLONG`
: Nazwy `oldpath` lub `newpath` są zbyt długie.

[5] Choć w Linuksie można zamontować urządzenie w wielu miejscach struktury katalogów, nie wolno wciąż zmienić nazwy jednego punktu montowania na inny, nawet jeśli oba znajdują się w tym samym urządzeniu.

ENOENT
: Element ścieżki `oldpath` lub `newpath` nie istnieje bądź jest zawieszonym dowiązaniem symbolicznym.

ENOMEM
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

ENOSPC
: W urządzeniu nie ma już wolnego miejsca, aby zakończyć tę operację.

ENOTDIR
: Element ścieżek `oldpath` lub `newpath` (oprócz ewentualnego elementu końcowego) nie jest katalogiem bądź `oldpath` jest katalogiem, natomiast `newpath` istnieje, lecz nim nie jest.

ENOTEMPTY
: Ścieżka `newpath` jest niepustym katalogiem.

EPERM
: Istnieje przynajmniej jedna ze ścieżek podanych w parametrach wejściowych, katalog bezpośrednio nadrzędny posiada ustawiony bit sticky (`S_ISVTX`), efektywny identyfikator użytkownika dla procesu wywołującego nie jest identyfikatorem użytkownika tego pliku ani katalogu, w którym się znajduje, a sam proces nie jest uprzywilejowany.

EROFS
: System plików posiada uprawnienia dla wykonywania operacji tylko do odczytu.

EXDEV
: Ścieżki `oldpath` i `newpath` znajdują się w różnych systemach plików.

Tabela 7.1. przedstawia wyniki przenoszenia różnych rodzajów plików.

Tabela 7.1. Wyniki przenoszenia różnych rodzajów plików

	Element docelowy jest plikiem	**Element docelowy jest katalogiem**	**Element docelowy jest dowiązaniem**	**Element docelowy nie istnieje**
Element źródłowy jest plikiem	Element docelowy zostanie nadpisany przez element źródłowy.	Błąd `EISDIR`.	Plik zostanie przeniesiony i element docelowy będzie nadpisany.	Plik zostanie przeniesiony.
Element źródłowy jest katalogiem	Błąd `ENOTDIR`.	Jeśli element docelowy jest pusty, wówczas nazwa elementu źródłowego zostanie zmieniona na nazwę elementu docelowego; w przeciwnym razie pojawi się błąd `ENOTEMPTY`.	Katalog zostanie przeniesiony, a element docelowy będzie nadpisany.	Katalog zostanie przeniesiony.
Element źródłowy jest dowiązaniem	Dowiązanie zostanie przeniesione, a element docelowy będzie nadpisany.	Błąd `EISDIR`.	Dowiązanie zostanie przeniesione, a element docelowy będzie nadpisany.	Dowiązanie zostanie przeniesione.
Element źródłowy nie istnieje	Błąd `ENOENT`.	Błąd `ENOENT`.	Błąd `ENOENT`.	Błąd `ENOENT`.

Jeśli element źródłowy i docelowy znajdują się w różnych systemach plików, wówczas dla wszystkich powyższych przypadków, bez względu na ich rodzaj, funkcja nie powiedzie się i zwróci błąd `EXDEV`.

Węzły urządzeń

Węzły urządzeń są plikami specjalnymi, które umożliwiają aplikacjom na korzystanie ze sterowników urządzeń. Gdy aplikacja, działająca w systemie Unix, wykonuje zwykłe operacje wejścia i wyjścia (otwieranie, zamykanie, czytanie, pisanie itd.) w węźle urządzenia, wówczas żądania te nie są obsługiwane przez jądro w standardowy sposób. Jądro przekazuje je do sterownika urządzenia. Sterownik urządzenia obsługuje operację wejścia i wyjścia oraz zwraca wynik jej wykonania do użytkownika. Węzły urządzeń dostarczają abstrakcji urządzenia, dlatego też podczas tworzenia aplikacji nie istnieje potrzeba posiadania dokładnych informacji na temat szczegółów technicznych działania urządzenia lub nawet głównych interfejsów specjalnych. Prawdą jest, że węzły urządzeń są standardowym mechanizmem pozwalającym na dostęp do sprzętu w systemach uniksowych. Urządzenia sieciowe są rzadkim odstępstwem od tej reguły, a mimo to w ciągu całej historii systemu Unix zdarzały się spory, w których pojawiały się argumenty, iż nawet ten wyjątek jest pomyłką. Obsługa wszystkich urządzeń w maszynie za pomocą funkcji `read()`, `write()` oraz `mmap()` jest rzeczywiście wygodnym rozwiązaniem.

W jaki sposób jądro rozpoznaje dany sterownik urządzenia, do którego powinno przekazać obsługę operacji? Każdemu węzłowi urządzenia przypisane są dwie liczbowe wartości, zwane *numerem głównym* (ang. *major number*) oraz *numerem drugorzędnym* (ang. *minor number*). Te dwie liczby odwzorowują urządzenie na określony sterownik załadowany do jądra. Jeśli węzeł urządzenia posiada oba numery nieodpowiadające sterownikowi załadowanemu do jądra (co czasem się zdarza), wówczas funkcja `open()`, wywołana dla węzła urządzenia, zwraca wartość `-1` oraz ustawia zmienną `errno` na `ENODEV`. Mówi się wówczas, że takie węzły odnoszą się do nieistniejących urządzeń.

Specjalne węzły urządzeń

Specjalne węzły urządzeń istnieją we wszystkich systemach linuksowych. Są częścią środowiska projektowego Linuksa, a ich obecność jest potwierdzona w interfejsie binarnym aplikacji (ABI) dla tego systemu.

Urządzenie puste (ang. *null device*) posiada numer główny równy liczbie 1, natomiast numer drugorzędny równy 3. Ścieżką dostępu do niego jest */dev/null*. Właścicielem tego urządzenia powinien być administrator. Musi mieć ono uprawnienia do zapisu i odczytu przez wszystkich użytkowników. Jądro domyślnie odrzuca wszystkie żądania zapisu do niego, natomiast żądania odczytu zwracają znacznik końca pliku (EOF).

Urządzenie zerowe (ang. *zero device*) znajduje się w */dev/zero* i posiada numer główny równy 1 oraz numer drugorzędny 5. Podobnie jak w przypadku urządzenia pustego, jądro domyślnie odrzuca wszystkie żądania zapisu do niego. Odczyt z urządzenia zerowego zwraca nieskończony ciąg bajtów zerowych.

Urządzenie zapełnione (ang. *full device*), posiadające numer główny równy 1 oraz numer drugorzędny 7, znajduje sie w */dev/full*. Podobnie jak w przypadku urządzenia zerowego, odczyt

z niego zwraca nieskończony ciąg bajtów zerowych (`\0`). Jednak zapisy do niego zawsze powodują powstanie błędu `ENOSPC`, który oznacza, że w danym urządzeniu nie ma już wolnego miejsca.

Powyżej opisane urządzenia zostały stworzone dla różnych celów. Są przydatne, aby sprawdzić, jak aplikacja reaguje na sytuacje ekstremalne i krytyczne — na przykład na zapełniony system plików. Ponieważ urządzenia puste i zerowe ignorują żądania zapisów, mogą również zapewnić prosty sposób na odrzucenie niechcianych operacji wejścia i wyjścia.

Generator liczb losowych

Urządzenia generatora liczb losowych jądra znajdują się w */dev/random* oraz */dev/urandom*. Posiadają numer główny równy 1, a numery drugorzędne równe odpowiednio 8 i 9.

Generator liczb losowych jądra odbiera zakłócenia ze sterowników urządzeń oraz innych źródeł, a jądro łączy je razem i jednocześnie jednokierunkowo koduje. Wynik tej operacji zapisany zostaje do *puli entropii* (ang. *entropy pool*). Jądro potrafi ocenić liczbę bitów entropii w tej puli.

Odczyt z urządzenia */dev/random* zwraca entropię z danej puli. Wyniki są wystarczające, by utworzyć początkową wartość (ziarno) dla generatorów liczb losowych, przeprowadzić generowanie klucza i inne operacje, które wymagają silnej entropii kryptograficznej.

Przeciwnik, który otrzymał wystarczająco dużo danych z puli entropii oraz z sukcesem potrafił złamać system kodowania jednokierunkowego, mógłby teoretycznie uzyskać wiedzę na temat stanu pozostałej części puli entropii. Mimo że taki atak jest obecnie tylko teoretyczną możliwością (nie są znane jawne wystąpienia takich przypadków), jądro uwzględnia go przez zmniejszanie wartości oszacowania ilości entropii w puli po wykonaniu każdej operacji odczytu. Jeśli oszacowanie osiągnie zero, odczyty z urządzenia */dev/random* zostaną zablokowane, dopóki system nie wygeneruje więcej entropii, a oszacowanie entropii nie stanie się na tyle duże, aby móc kontynuować operacje odczytów.

Urządzenie */dev/urandom* nie posiada tej właściwości; odczyt z niego powiedzie się nawet wtedy, gdy oszacowanie entropii jądra będzie niewystarczające, aby zakończyć to żądanie. Ponieważ tylko aplikacje, które wymagają zapewnienia najwyższego bezpieczeństwa (takie jak generowanie kluczy dla celów wymiany danych w programie GNU Privacy Guard), mogłyby być zainteresowane w wykorzystaniu silnej entropii kryptograficznej, dlatego też większość programów powinna używać */dev/urandom* zamiast */dev/random*. Odczyt z tego drugiego urządzenia mógłby się ewentualnie zablokować na dłuższy czas, jeśli nie istniałyby aktywne operacje wejścia i wyjścia, które zasilałyby pulę entropii jądra. Nie jest to rzadkie zjawisko w przypadku serwerów bez dysków i głowic.

Komunikacja poza kolejką

Model plików systemu Unix jest imponujący. Za pomocą prostych operacji odczytu i zapisu system Unix tworzy abstrakcje praktycznie każdej możliwej do wyobrażenia sobie czynności, którą można by przeprowadzić dla obiektów. Czasami jednak programiści w swoich aplikacjach wymagają połączenia się z plikiem znajdującym sie poza głównym obszarem obsługi danych. Rozważmy urządzenie portu szeregowego. Czytanie z tego urządzenia spowodowałoby odczyt ze sprzętu podłączonego do drugiej końcówki kabla szeregowego, natomiast zapis

do niego zaowocowałby wysłaniem danych również do tego samego sprzętu. W jaki sposób proces mógłby jednak odczytać dane z jednego ze specjalnych styków stanu portu szeregowego, takich jak sygnał gotowości terminala (DTR)? Względnie, w jaki sposób mógłby ustawić parzystość portu szeregowego?

Odpowiedzią jest użycie funkcji `ioctl()`. Funkcja `ioctl()`, której nazwa oznacza *sterowanie operacjami wejścia i wyjścia* (ang. *I/O control*), pozwala na komunikację poza kolejką:

```
#include <sys/ioctl.h>

int ioctl (int fd, int request, ...);
```

Wymaga dwóch parametrów. Oto one:

`fd`
: Deskryptor pliku.

`request`
: Specjalna wartość kodu dla żądania, które zostało wcześniej zdefiniowane i zaakceptowane przez proces i jądro systemu. Kod ten opisuje operację, która powinna zostać wykonana dla pliku reprezentowanego przez deskryptor `fd`.

Funkcja `ioctl()` może również używać jednego lub więcej nietypowych parametrów opcjonalnych (zwykle będących liczbami całkowitymi bez znaku lub wskaźnikami), które są przekazywane do jądra.

Poniższy program używa kodu żądania `CDROMEJECT`, aby wysunąć tackę z napędu CD-ROM. Kod ten zostaje przekazany przez użytkownika jako pierwszy argument wywołania programu. Stąd też program ten działa podobnie jak standardowe polecenie *eject*:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/cdrom.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
   int fd, ret;

   if (argc < 2)
   {
      fprintf (stderr, "Użycie programu: %s <urządzenie do wysunięcia>\n", argv[0]);
      return 1;
   }

  /*
  * Otwiera urządzenie CD-ROM w trybie tylko do odczytu.
  * Znacznik O_NONBLOCK informuje jądro, że urządzenie należy
  * otworzyć nawet wówczas, gdy nie zawiera w sobie nośnika danych.
  */
   fd = open (argv[1], O_RDONLY | O_NONBLOCK);
   if (fd < 0)
   {
      perror ("open");
      return 1;
   }
```

```
    /* Wysłanie polecenia wysunięcia tacki do urządzenia CD-ROM. */
    ret = ioctl (fd, CDROMEJECT, 0);
    if (ret)
    {
       perror ("ioctl");
       return 1;
    }

    ret = close (fd);
    if (ret)
    {
       perror ("close");
       return 1;
    }

    return 0;
}
```

Kod żądania `CDROMEJECT` jest jedną z opcji sterownika urządzenia CD-ROM w Linuksie. Gdy jądro otrzymuje żądanie `ioctl()`, wówczas szuka systemu plików (w przypadku plików rzeczywistych) lub sterownika urządzenia (w przypadku węzłów urządzeń), zawierającego deskryptor pliku dostarczony do funkcji, a następnie przekazuje mu żądanie do obsługi. W tym przypadku żądanie odbiera sterownik urządzenia CD-ROM, wysuwając tackę z napędu.

W dalszej części rozdziału przedstawiony zostanie przykład wykorzystania funkcji `ioctl()`, dla której użyty będzie opcjonalny parametr w celu zwrócenia informacji do wywołującego ją procesu.

Śledzenie zdarzeń związanych z plikami

Linux udostępnia interfejs, zwany *inotify*, który pozwala na monitorowanie plików — na przykład, aby otrzymać odpowiednie powiadomienie, gdy są one przenoszone, odczytywane, zapisywane lub usuwane. Załóżmy, że należy stworzyć oprogramowanie graficznego zarządcy plików, takiego jak Nautilus działający w środowisku GNOME. Jeśli plik zostanie skopiowany do katalogu, którego zawartość wyświetla Nautilus, wówczas widok, udostępniany przez zarządcę plików, stanie się niespójny.

Jednym z rozwiązań tego problemu jest ciągłe odczytywanie zawartości katalogu, a przez to wykrywanie zmian i aktualizowanie widoku. Rozwiązanie to powoduje jednak okresowe obciążenie systemu i jest dalekie od ideału. Co gorsze, istnieje ciągłe współzawodnictwo pomiędzy operacją usuwania lub dodawania plików do katalogu, a operacją odświeżania jego widoku.

Dzięki interfejsowi *inotify*, jądro może *przekazać* dane zdarzenie do aplikacji w momencie, gdy ma ono miejsce. Gdy tylko plik zostaje skasowany, Nautilus otrzymuje o tym informację od jądra. W odpowiedzi może natychmiast usunąć skasowany plik z widoku przedstawiającego w graficzny sposób zawartość katalogu.

Zdarzenia związane z plikami są również ważne dla wielu innych aplikacji. Rozważmy program użytkowy, służący do wykonywania kopii zapasowych lub narzędzie tworzące indeksy w bazie danych. Interfejs *inotify* pozwala obu tym programom działać w czasie rzeczywistym: w tym samym momencie, gdy pliki zostają stworzone, usunięte lub zapisane, programy narzędziowe mogą uaktualnić archiwum kopii zapasowych lub indeks bazy danych.

Interfejs *inotify* zastąpił interfejs *dnotify* będący wcześniejszą wersją mechanizmu monitorującego pliki i posiadającego skomplikowaną obsługę opartą na sygnałach. W aplikacjach należy

zawsze używać *inotify* zamiast *dnotify*. Interfejs *inotify*, wprowadzony w wersji jądra 2.6.13, jest uniwersalny i łatwy w użyciu, ponieważ te same operacje, które programy wykonują w przypadku plików zwykłych, szczególnie takie jak `select()` i `poll()`, są możliwe do przeprowadzenia w nim. W tej książce omówiony zostanie tylko interfejs *inotify*.

Inicjalizacja interfejsu inotify

Zanim interfejs *inotify* będzie mógł zostać użyty przez proces, musi być zainicjalizowany. Funkcja systemowa `inotify_init()` pozwala na zainicjalizowanie tego interfejsu i zwraca deskryptor pliku reprezentujący jego zainicjalizowany egzemplarz:

```
#include <inotify.h>

int inotify_init (void);
```

W przypadku błędu, funkcja `inotify_init()` zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EMFILE`
: Osiągnięto maksymalną liczbę egzemplarzy interfejsu *inotify*, które może jednocześnie używać użytkownik.

`ENFILE`
: Osiągnięto systemowe ograniczenie maksymalnej dopuszczalnej liczby deskryptorów plików.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

Poniższy kod wykonuje operację inicjalizowania interfejsu *inotify*, dlatego też będzie możliwe jego użycie w następnych etapach:

```
int fd;

fd = inotify_init ( );
if (fd == -1)
{
   perror ("inotify_init");
   exit (EXIT_FAILURE);
}
```

Elementy obserwowane

Po zainicjalizowaniu interfejsu *inotify*, proces może skonfigurować *elementy obserwowane* (ang. *watches*). Element obserwowany, reprezentowany przez *deskryptor elementu obserwowanego*, jest standardową ścieżką Uniksa, a także posiada skojarzoną *maskę elementu obserwowanego*, która informuje jądro, jakiego typu zdarzenia powinien obserwować proces — na przykład tylko odczyty, tylko zapisy lub odczyty, jak i zapisy.

Interfejs *inotify* monitoruje zarówno pliki, jak i katalogi. W przypadku obserwowania katalogów, *inotify* informuje o zdarzeniach dotyczących samego katalogu, jak również plików, które się w nim znajdują (nie dotyczy to plików znajdujących się w podkatalogach obserwowanego katalogu — operacja monitorowania nie jest rekurencyjna).

Dodawanie nowego elementu obserwowanego

Funkcja systemowa `inotify_add_watch()` dodaje nowy element obserwowany dla zdarzenia lub zdarzeń, opisanych przez parametr `mask`, dotyczących pliku lub katalogu `path`. Element ten zostaje dodany do egzemplarza interfejsu *inotify*, reprezentowanego przez deskryptor `fd`:

```
#include <inotify.h>

int inotify_add_watch (int fd, const char *path, uint32_t mask);
```

W przypadku sukcesu funkcja zwraca nowy deskryptor elementu obserwowanego. W przypadku błędu funkcja `inotify_add_watch()` zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EACCESS`
: Plik, wskazywany przez ścieżkę `path`, nie posiada uprawnień do odczytu. Proces wywołujący musi mieć możliwość czytania z pliku, aby dodać dla niego element obserwowany.

`EBADF`
: Deskryptor pliku `fd` nie jest poprawnym egzemplarzem interfejsu *inotify*.

`EFAULT`
: Wskaźnik `path` jest niepoprawny.

`EINVAL`
: Maska elementu obserwowanego, reprezentowana przez parametr `mask`, zawiera niepoprawne zdarzenia.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby zakończyć tę operację.

`ENOSPC`
: Osiągnięto maksymalną liczbę egzemplarzy interfejsu *inotify*, które może jednocześnie używać użytkownik.

Maska elementu obserwowanego

Maska elementu obserwowanego jest sumą binarną jednego lub więcej zdarzeń interfejsu *inotify*, które są zdefiniowane w pliku nagłówkowym `<inotify.h>`:

`IN_ACCESS`
: Wystąpił odczyt z pliku.

`IN_MODIFY`
: Wystąpił zapis do pliku.

`IN_ATTRIB`
: Metadane pliku (na przykład właściciel, uprawnienia lub atrybuty rozszerzone) zostały zmienione.

`IN_CLOSE_WRITE`
: Plik został zamknięty, będąc otwartym do zapisu.

`IN_CLOSE_NOWRITE`
: Plik został zamknięty, nie będąc otwartym do zapisu.

`IN_OPEN`
: Plik został otwarty.

`IN_MOVED_FROM`
: Plik został przeniesiony z obserwowanego katalogu.

`IN_MOVED_TO`
: Plik został przeniesiony do obserwowanego katalogu.

`IN_CREATE`
: Plik został stworzony w obserwowanym katalogu.

`IN_DELETE`
: Plik został usunięty z obserwowanego katalogu.

`IN_DELETE_SELF`
: Obiekt obserwowany usunął się samoczynnie.

`IN_MOVE_SELF`
: Obiekt obserwowany przeniósł się samoczynnie.

Następujące znaczniki są również zdefiniowane, łącząc dwa zdarzenia lub więcej w jedną całość:

`IN_ALL_EVENTS`
: Wszystkie dozwolone zdarzenia.

`IN_CLOSE`
: Wszystkie zdarzenia związane z zamykaniem (aktualnie `IN_CLOSE_WRITE` oraz `IN_CLOSE_NOWRITE`).

`IN_MOVE`
: Wszystkie zdarzenia związane z przenoszeniem (aktualnie `IN_MOVED_FROM` oraz `IN_MOVED_TO`).

Poniższy kod dodaje nowy element obserwowany do istniejącego egzemplarza interfejsu *inotify*:

```
int wd;

wd = inotify_add_watch (fd, "/etc", IN_ACCESS | IN_MODIFY);
if (wd == -1)
{
   perror ("inotify_add_watch");
   exit (EXIT_FAILURE);
}
```

Przykład ten dodaje element obserwowany, monitorujący wszystkie zapisy i odczyty występujące w katalogu */etc*. Jeśli dowolny plik w tym katalogu zostanie odczytany lub zapisany, interfejs *inotify* wyśle zdarzenie do deskryptora pliku tego interfejsu `fd` udostępniającego deskryptor elementu obserwowanego `wd`. W dalszej części tego rozdziału zostanie wyjaśnione, w jaki sposób interfejs *inotify* reprezentuje zdarzenia.

Zdarzenia interfejsu inotify

Struktura `inotify_event`, zdefiniowana w pliku nagłówkowym `<inotify.h>`, reprezentuje zdarzenia interfejsu *inotify*:

```
#include <inotify.h>

struct inotify_event
{
   int wd; /* deskryptor elementu obserwowanego */
   uint32_t mask; /* maska zdarzeń */
```

```
    uint32_t cookie; /* unikalny znacznik kontekstu */
    uint32_t len; /* rozmiar pola 'name' */
    char name[]; /* nazwa, zakończona znakiem pustym */
};
```

Pole `wd` oznacza deskryptor elementu obserwowanego, który otrzymano po wywołaniu funkcji `inotify_add_watch()`, a pole `mask` reprezentuje zdarzenia. Jeśli parametr `wd` określa katalog, w którym dla pewnego pliku wystąpiło jedno z obserwowanych zdarzeń, wówczas parametr `name` oznacza nazwę tego pliku, względną w stosunku do wspomnianego katalogu. W tym przypadku parametr `len` jest wartością niezerową. Należy zauważyć, że parametr `len` nie jest równy długości nazwy `name`; łańcuch `name` może posiadać więcej znaków pustych niż tylko ten jeden, który służy jako jego zakończenie. Dodatkowe znaki puste spełniają rolę wypełnienia, aby zapewnić, że kolejna struktura `inotify_event` zostanie odpowiednio wyrównana. Zgodnie z tym, aby obliczyć wartość przesunięcia do następnej struktury `inotify_event` znajdującej się w tablicy, należy użyć pola `len`, a nie funkcji `strlen()`.

Na przykład, jeśli parametr `wd` reprezentuje */home/rlove*, odpowiadająca mu wartość `mask` wynosi `IN_ACCESS`, a dla pliku */home/rlove/canon* zostanie wykonana operacja odczytu, wówczas pole `name` będzie równe łańcuchowi znaków `canon`, a wartość `len` będzie wynosić co najmniej `6`. I odwrotnie, jeśli obserwowany byłby bezpośrednio sam plik */home/rlove/canon* z poprzednio użytą wartością maski zdarzeń, wówczas pole `len` równałoby się zeru, a ciąg znaków `name` miałby długość zerową i nie istniałaby potrzeba, aby go modyfikować.

Pole `cookie` używane jest, aby połączyć dwa związane ze sobą, lecz rozłączne zdarzenia. Zostanie ono omówione w następnym podrozdziale.

Odczytywanie zdarzeń inotify

Otrzymywanie zdarzeń *inotify* jest łatwe: należy po prostu odczytywać deskryptor pliku, związany z egzemplarzem interfejsu *inotify*. Interfejs ten dostarcza opcji, zwanej *zasysaniem* (ang. *slurping*), która pozwala na odczytywanie wielu zdarzeń (tylu, ile zmieści się w buforze dostarczonym do funkcji `read()`) przy użyciu pojedynczego żądania odczytu. Z powodu zmiennej długości pola `name`, jest to najbardziej powszechny sposób na odczytywanie zdarzeń *inotify*.

Poprzednie przykłady stworzyły egzemplarz interfejsu *inotify* oraz dodały do niego element obserwowany. Obecnie przedstawiony zostanie kod odczytujący oczekujące zdarzenia:

```
char buf[BUF_LEN]_attribute_((aligned(4)));
ssize_t len, i = 0;

/* odczytać ciąg bajtów o długości BUF_LEN, dotyczących zdarzeń */
len = read (fd, buf, BUF_LEN);

/* wykonać iterację dla każdego zdarzenia czytania, dopóki jeszcze jakieś istnieją */
while (i < len)
{
   struct inotify_event *event = (struct inotify_event *) &buf[i];
   printf ("wd=%d mask=%d cookie=%d len=%d katalog=%s\n", event->wd, event->mask,
     event->cookie, event->len, (event->mask & IN_ISDIR) ? "tak" : "nie");

   /* jeśli parametr 'name' istnieje, należy go wyprowadzić */
   if (event->len)
      printf ("nazwa=%s\n", event->name);

   /* zaktualizować indeks, aby uzyskać dostęp do następnego zdarzenia */
   i += sizeof (struct inotify_event) + event->len;
}
```

Ponieważ deskryptor pliku dla interfejsu *inotify* zachowuje się jak plik zwykły, można go obserwować w programach za pomocą funkcji `select()`, `poll()` i `epoll()`. Pozwala to procesom na monitorowanie zdarzeń *inotify* w pojedynczym wątku dla operacji wejścia i wyjścia, występujących w różnych plikach.

Zaawansowane zdarzenia inotify. Jako dodatek do zdarzeń standardowych, interfejs *inotify* może generować również inne zdarzenia:

`IN_IGNORED`
: Element obserwowany, reprezentowany przez `wd`, został usunięty. Zdarzenie to może wystąpić, gdy użytkownik ręcznie usunął element obserwowany lub gdy obserwowany obiekt już nie istnieje. Zostanie ono omówione w następnym podrozdziale.

`IN_ISDIR`
: Używany obiekt jest katalogiem (jeśli zdarzenie to nie jest ustawione, używany obiekt jest plikiem).

`IN_Q_OVERFLOW`

Kolejka interfejsu *inotify* została przepełniona. Jądro ogranicza rozmiar kolejki zdarzeń, aby zabezpieczyć się przed nieograniczonym zużyciem pamięci. Gdy tylko liczba oczekujących zdarzeń osiąga wartość o jeden mniejszą od dopuszczalnego maksimum, jądro generuje zdarzenie `IN_Q_OVERFLOW` i dołącza go na koniec kolejki. Dopóki kolejka nie zostanie odczytana, nie będą generowane następne zdarzenia, dzięki czemu jej rozmiar nie przekroczy maksymalnego limitu.

`IN_UNMOUNT`

Urządzenie, w którym znajdował się obserwowany obiekt, zostało odmontowane. Dlatego też sam obiekt nie jest już dostępny; jądro usunie odpowiedni element obserwowany i wygeneruje zdarzenie `IN_UNMOUNT`.

Powyższe zdarzenia mogą zostać wygenerowane przez dowolny element obserwowany; użytkownik nie musi ich jawnie ustawiać.

Programiści powinni używać pola `mask` jako maski bitów zdarzeń oczekujących. Zgodnie z tym *nie można* sprawdzać zdarzeń, używając bezpośredniego operatora porównania:

```
/* Tak NIE WOLNO robić! */
if (event->mask == IN_MODIFY)
   printf ("Wystąpił zapis do pliku!\n");
else if (event->mask == IN_Q_OVERFLOW)
   printf ("Uwaga, przepełnienie kolejki!\n);
```

Należy przeprowadzić natomiast testy bitowe:

```
if (event->mask & IN_ACCESS)
   printf ("Wystąpił zapis do pliku!\n");
if (event->mask & IN_UNMOUNTED)
   printf ("Urządzenie, w którym znajduje się plik, zostało odmontowane!\n);
if (event->mask & IN_ISDIR)
   printf ("Plik jest katalogiem!\n");
```

Łączenie zdarzeń przenoszenia

Każde ze zdarzeń `IN_MOVED_FROM` oraz `IN_MOVED_TO` reprezentuje połowę operacji przenoszenia pliku: pierwsze z nich dotyczy usunięcia pliku z poprzedniego położenia, drugie reprezentuje jego pojawienie się w nowej lokalizacji. Dlatego też, jeśli procesy mają być rzeczywiście

przydatne w programie zamierzającym inteligentnie śledzić ruch plików w systemie (weźmy pod uwagę program indeksujący, który nie potrafi stworzyć indeksu dla przeniesionego pliku), muszą mieć one możliwość połączenia tych dwóch wspomnianych zdarzeń.

Pole `cookie` ze struktury `inotify_event` pozwala rozwiązać ten problem.

Gdy pole to jest niezerowe, zawiera wówczas wartość łączącą ze sobą dwa zdarzenia. Rozważmy proces, który obserwuje katalogi */bin* oraz */sbin*. Załóżmy, że */bin* posiada deskryptor elementu obserwowanego równy `7`, natomiast */sbin* posiada również taki deskryptor, ale o wartości `8`. Jeśli plik */bin/compass* zostanie przeniesiony na miejsce */sbin/compass*, jądro wygeneruje dwa zdarzenia *inotify*.

Pierwsze zdarzenie będzie posiadać pole `wd` równe `7`, maskę `mask` równą `IN_MOVED_FROM` oraz nazwę `name` równą łańcuchowi `compass`. Drugie zdarzenie będzie posiadać pole `wd` równe `8`, maskę `mask` równą `IN_MOVED_TO` oraz nazwę `name` zawierająca również łańcuch `compass`. Dla obu tych zdarzeń pole `cookie` będzie takie samo i równe na przykład `12`.

Jeśli plik zmienia nazwę, jądro wciąż generuje dwa zdarzenia. Pole `wd` jest wówczas dla nich identyczne.

Należy zauważyć, że jeśli plik jest przenoszony z katalogu lub do katalogu, który nie jest obserwowany, proces nie otrzyma jednego z odpowiednich zdarzeń. Wyłącznie od programu zależy, czy zareaguje on na to, że drugie zdarzenie, posiadające taką samą wartość pola `cookie`, nigdy się nie pojawi.

Zaawansowane opcje obserwowania

Podczas tworzenia nowego elementu obserwowanego do pola `mask` można dodać jeden lub więcej znaczników, aby zmodyfikować zachowanie tego elementu:

`IN_DONT_FOLLOW`
: Jeśli ta wartość jest ustawiona, a element docelowy ścieżki `path` lub dowolny jego komponent jest dowiązaniem symbolicznym, wówczas nie będzie on użyty i wywołanie funkcji `inotify_add_watch()` nie powiedzie się.

`IN_MASK_ADD`
: Zwykle, jeśli funkcja `inotify_add_watch()`używana jest dla pliku, dla którego istnieje element obserwowany, maska tego elementu zostaje zmodyfikowana, aby odzwierciedlić aktualną sytuację i nową wartość pola `mask`. Gdy znacznik `IN_MASK_ADD` jest ustawiony w tym polu, wówczas nowo pojawiające się zdarzenia zostaną dodane do istniejącej maski.

`IN_ONESHOT`
: Jeśli wartość ta jest ustawiona, jądro automatycznie usunie element obserwowany po pojawieniu się pierwszego zdarzenia związanego z danym obiektem. Element ten będzie obserwowany jednorazowo.

`IN_ONLYDIR`
: Jeśli wartość ta jest ustawiona, wówczas element obserwowany zostanie dodany wyłącznie wtedy, gdy dostarczonym obiektem będzie katalog. Jeśli ścieżka `path` reprezentuje plik, a nie katalog, wówczas wywołanie funkcji `inotify_add_watch()` się nie powiedzie.

Na przykład, poniższy kod dodaje element obserwowany dla ścieżki */etc/init.d* jedynie wtedy, gdy *init.d* jest katalogiem, a jednocześnie ani */etc*, ani */etc/init.d* nie są dowiązaniami symbolicznymi:

```
int wd;

/*
* Obserwuj obiekt '/etc/init.d', aby śledzić operacje jego przenoszenia,
* lecz jedynie wówczas, gdy jest on katalogiem i żaden element jego ścieżki
* nie jest dowiązaniem symbolicznym.
*/
wd = inotify_add_watch (fd, "/etc/init.d", IN_MOVE_SELF | IN_ONLYDIR |
IN_DONT_FOLLOW);
if (wd == -1)
   perror ("inotify_add_watch");
```

Usuwanie elementu obserwowanego z interfejsu inotify

Element obserwowany może zostać usunięty z egzemplarza interfejsu *inotify* za pomocą funkcji systemowej `inotify_rm_watch()`:

```
#include <inotify.h>

int inotify_rm_watch (int fd, uint32_t wd);
```

Poprawne wywołanie funkcji `inotify_rm_watch()` usuwa element obserwowany, reprezentowany przez deskryptor elementu obserwowanego `wd`, z egzemplarza interfejsu *inotify* (reprezentowanego przez deskryptor pliku) `fd`, a następnie zwraca `0`.

Na przykład:

```
int ret;

ret = inotify_rm_watch (fd, wd);
if (ret)
   perror ("inotify_rm_watch");
```

W przypadku błędu, funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EBADF`
: Parametr `fd` nie jest poprawnym egzemplarzem interfejsu *inotify*.

`EINVAL`
: Parametr `wd` nie jest poprawnym deskryptorem elementu obserwowanego dla danego egzemplarza interfejsu *inotify*.

Podczas usuwania elementu obserwowanego, jądro generuje zdarzenie `IN_IGNORED`. Wysyłane zostaje ono nie tylko podczas ręcznego usunięcia, lecz również wtedy, gdy jest efektem ubocznym wykonania innej operacji. Na przykład, gdy obserwowany plik zostanie skasowany, wszystkie jego elementy obserwowane również będą usunięte. W takich przypadkach jądro wysyła zdarzenie `IN_IGNORED`. Zachowanie to pozwala aplikacjom na usuwanie elementów obserwowanych w jednym miejscu, czyli w procedurze obsługi zdarzenia `IN_IGNORED`. Jest to użyteczne dla zaawansowanych użytkowników interfejsu *inotify*, którzy zarządzają złożonymi strukturami danych, kontrolowanymi przez odpowiednie elementy obserwowane. Przykładem takiego rozwiązania jest infrastruktura przeszukiwania w aplikacji Beagle, działającej w środowisku GNOME.

Otrzymywanie rozmiaru kolejki zdarzeń

Rozmiar kolejki zdarzeń oczekujących może zostać ustalony za pomocą opcji sterującej operacjami wejścia i wyjścia, równej `FIONREAD` i użytej dla deskryptora pliku egzemplarza interfejsu *inotify*. Rozmiar kolejki, wyrażony w bajtach i będący liczbą całkowitą bez znaku, zostaje zapisany do pierwszego parametru żądania:

```
unsigned int queue_len;
int ret;

ret = ioctl (fd, FIONREAD, &queue_len);
if (ret < 0)
   perror ("ioctl");
else
   printf ("%u bajtów oczekuje w kolejce\n", queue_len);
```

Należy zauważyć, że funkcja zwraca rozmiar kolejki w bajtach, a nie liczbę zdarzeń w kolejce. Program może oszacować liczbę zdarzeń, biorąc pod uwagę liczbę bajtów i używając znanego rozmiaru struktury `inotify_event` (uzyskanego za pomocą funkcji `sizeof()`), jednocześnie zakładając przeciętny rozmiar pola `name`. Bardziej użyteczne jest jednak to, że liczba oczekujących bajtów jest po prostu wymaganym rozmiarem pakietu danych, który powinien zostać odczytany przez proces.

Stała `FIONREAD` jest zdefiniowana w pliku nagłówkowym `<sys/ioctl.h>`.

Usuwanie egzemplarza interfejsu inotify

Usuwanie egzemplarza interfejsu *inotify* oraz związanych z nim elementów obserwowanych jest proste, ponieważ polega na zamknięciu deskryptora pliku dla tego egzemplarza:

```
int ret;

/* deskryptor 'fd' otrzymano za pomocą funkcji inotify_init() */
ret = close (fd);
if (fd == -1)
   perror ("close");
```

Oczywiście, podobnie jak dla każdego deskryptora pliku, będzie on również zamknięty, a jego zasoby „posprzątane”, gdy proces zostanie zakończony.

ROZDZIAŁ 8.

Zarządzanie pamięcią

Pamięć należy do najbardziej podstawowych, a jednocześnie najważniejszych zasobów dostępnych dla procesu. W rozdziale tym omówione zostaną tematy związane z zarządzaniem nią: przydzielanie, modyfikowanie i w końcu zwalnianie pamięci.

Słowo *przydzielanie* — powszechnie używany termin, określający czynność udostępniania obszaru pamięci — wprowadza w błąd, ponieważ wywołuje obraz wydzielania deficytowego zasobu, dla którego wielkość żądań przewyższa wielkość zapasów. Na pewno wielu użytkowników wolałoby mieć więcej dostępnej pamięci. Dla nowoczesnych systemów problem nie polega jednak na rozdzielaniu zbyt małych zapasów dla zbyt wielu użytkowników, lecz właściwym używaniu i monitorowaniu danego zasobu.

W tym rozdziale przeanalizowane zostaną wszystkie metody przydzielania pamięci dla różnych obszarów programu, jednocześnie z ukazaniem ich zalet i wad. Przedstawimy również pewne sposoby, pozwalające na ustawianie i modyfikację zawartości dowolnych obszarów pamięci, a także wyjaśnimy, w jaki sposób należy zablokować dane w pamięci, aby w programach nie trzeba było oczekiwać na operacje jądra, które zajmowałoby się przerzucaniem danych z obszaru wymiany.

Przestrzeń adresowa procesu

Linux, podobnie jak inne nowoczesne systemy operacyjne, wirtualizuje swój fizyczny zasób pamięci. Procesy nie adresują bezpośrednio pamięci fizycznej. Zamiast tego jądro wiąże każdy proces z unikalną *wirtualną przestrzenią adresową*. Jest ona liniowa, a jej adresacja rozpoczyna się od zera i wzrasta do pewnej granicznej wartości maksymalnej.

Strony i stronicowanie

Wirtualna przestrzeń adresowa składa się ze *stron*. Architektura systemu oraz rodzaj maszyny determinują rozmiar strony, który jest stały: typowymi wartościami są na przykład 4 kB (dla systemów 32-bitowych) oraz 8 kB (dla systemów 64-bitowych)[1]. Strony są albo prawidłowe,

[1] Czasami systemy wspierają rozmiary stron, które mieszczą się w pewnym zakresie. Z tego powodu rozmiar strony nie jest częścią interfejsu binarnego aplikacji (ABI). Aplikacje muszą w sposób programowy uzyskać rozmiar strony w czasie wykonania. Zostało to opisane w rozdziale 4. i będzie jednym z tematów poruszonych w tym rozdziale.

albo nieprawidłowe. *Strona prawidłowa* (ang. *valid page*) związana jest ze stroną w pamięci fizycznej lub jakąś dodatkową pamięcią pomocniczą, np. partycją wymiany lub plikiem na dysku. *Strona nieprawidłowa* (ang. *invalid page*) nie jest z niczym związana i reprezentuje nieużywany i nieprzydzielony obszar przestrzeni adresowej. Dostęp do takiej strony spowoduje błąd segmentacji. Przestrzeń adresowa nie musi być koniecznie ciągła. Mimo że jest ona adresowana w sposób liniowy, zawiera jednak mnóstwo przerw, nieposiadających adresacji.

Program nie może użyć strony, która znajduje się w dodatkowej pamięci pomocniczej zamiast w fizycznej. Będzie to możliwe dopiero wtedy, gdy zostanie ona połączona ze stroną w pamięci fizycznej. Gdy proces próbuje uzyskać dostęp do adresu z takiej strony, układ zarządzania pamięcią (MMU) generuje *błąd strony* (ang. *page fault*). Wówczas wkracza do akcji jądro, w sposób niewidoczny *przerzucając* żądaną stronę z pamięci pomocniczej do pamięci fizycznej. Ponieważ istnieje dużo więcej pamięci wirtualnej niż rzeczywistej (nawet w przypadku systemów z pojedynczą wirtualną przestrzenią adresową!), jądro również przez cały czas *wyrzuca* strony z pamięci fizycznej do dodatkowej pamięci pomocniczej, aby zrobić miejsce na nowe strony, przerzucane w drugim kierunku. Jądro przystępuje do wyrzucania danych, dla których istnieje najmniejsze prawdopodobieństwo, iż będą użyte w najbliższej przyszłości. Dzięki temu następuje poprawa wydajności.

Współdzielenie i kopiowanie podczas zapisu

Wiele stron w pamięci wirtualnej, a nawet w różnych wirtualnych przestrzeniach adresowych należących do oddzielnych procesów, może być odwzorowanych na pojedynczą stronę fizyczną. Pozwala to różnym wirtualnym przestrzeniom adresowym na *współdzielenie* danych w pamięci fizycznej. Współdzielone dane mogą posiadać uprawnienia tylko do odczytu lub zarówno do odczytu, jak i zapisu.

Gdy proces przeprowadza operację zapisu do współdzielonej strony, posiadającej uprawnienia do wykonania tej czynności, mogą zaistnieć jedna lub dwie sytuacje. Najprostsza wersja polega na tym, że jądro zezwoli na wykonanie zapisu i wówczas wszystkie procesy współdzielące daną stronę, będą mogły „zobaczyć" wyniki tej operacji. Zezwolenie wielu procesom na czytanie lub zapis do współdzielonej strony wymaga zazwyczaj zapewnienia pewnego poziomu współpracy i synchronizacji między nimi.

Inaczej jest jednak, gdy układ zarządzania pamięcią przechwyci operację zapisu i wygeneruje wyjątek; w odpowiedzi, jądro w sposób niewidoczny stworzy nową kopię strony dla procesu zapisującego i zezwoli dla niej na kontynuowanie zapisu. To rozwiązanie zwane jest *kopiowaniem podczas zapisu* (ang. *copy-on-write*, w skrócie *COW*)[2]. Proces faktycznie posiada uprawnienia do odczytu dla współdzielonych danych, co przyczynia się do oszczędzania pamięci. Gdy proces chce zapisać do współdzielonej strony, otrzymuje wówczas na bieżąco unikalną jej kopię. Dzięki temu jądro może działać w taki sposób, jak gdyby proces zawsze posiadał swoją własną kopię strony. Ponieważ kopiowanie podczas zapisu jest zaimplementowane dla każdej strony z osobna, dlatego też duży plik może zostać efektywnie udostępniony wielu procesom, którym zostaną przydzielone unikalne strony fizyczne tylko wówczas, gdy będą chciały coś w nich zapisywać.

[2] W rozdziale 5. napisano, że funkcja `fork()` używa metody kopiowania podczas zapisu, aby powielić i udostępnić przestrzeń adresową rodzica tworzonemu procesowi potomnemu.

Regiony pamięci

Jądro rozmieszcza strony w blokach, które posiadają pewne wspólne cechy charakterystyczne, takie jak uprawnienia dostępu. Bloki te zwane są *regionami pamięci* (ang. *memory regions*), *segmentami* (ang. *segments*) lub *odwzorowaniami* (ang. *mappings*). Pewne rodzaje regionów pamięci mogą istnieć w każdym procesie:

- *Segment tekstu* zawiera kod programu dla danego procesu, literały łańcuchowe, stałe oraz inne dane tylko do odczytu. W systemie Linux segment ten posiada uprawnienia tylko do odczytu i jest odwzorowany bezpośrednio na plik obiektowy (program wykonywalny lub bibliotekę).
- *Segment stosu*, jak sama nazwa wskazuje, zawiera stos wykonania procesu. Segment stosu rozrasta się i maleje w sposób dynamiczny, zgodnie ze zmianami struktury stosu. Stos wykonania zawiera lokalne zmienne oraz dane zwracane z funkcji.
- *Segment danych* (lub *sterta*) zawiera dynamiczną pamięć procesu. Do segmentu tego można zapisaywać, a jego rozmiar się zmienia. Sterta jest zwracana przy użyciu funkcji `malloc()` (omówionej w następnym podrozdziale).
- *Segment bss*[3] zawiera niezainicjalizowane zmienne globalne. Zmienne te mają specjalne wartości (w zasadzie same zera), zgodnie ze standardem języka C. Linux optymalizuje je przy użyciu dwóch metod. Po pierwsze, ponieważ segment bss przeznaczony jest dla przechowywania niezainicjalizowanych danych, więc linker (*ld*) w rzeczywistości nie zapisuje specjalnych wartości do pliku obiektowego. Powoduje to zmniejszenie rozmiaru pliku binarnego. Po drugie, gdy segment ten zostaje załadowany do pamięci, jądro po prostu odwzorowuje go w trybie kopiowania podczas zapisu na stronę zawierającą same zera, co efektywnie ustawia domyślne wartości w zmiennych.
- Większość przestrzeni adresowej zajmuje grupa *plików odwzorowanych*, takich jak sam program wykonywalny, różne biblioteki — między innymi dla języka C, a także pliki z danymi. Ścieżka */proc/self/maps* lub wynik działania programu *pmap* są bardzo dobrymi przykładami plików odwzorowanych w procesie.

Rozdział ten omawia interfejsy, które są udostępnione przez system Linux, aby otrzymywać i zwalniać obszary pamięci, tworzyć i usuwać nowe odwzorowania oraz wykonywać inne czynności związane z pamięcią.

Przydzielanie pamięci dynamicznej

Pamięć zawiera także automatyczne i statyczne zmienne, lecz podstawą działania każdego systemu, który nią zarządza, jest przydzielanie, używanie, a w końcu zwalnianie *pamięci dynamicznej*. Pamięć dynamiczna przydzielana jest w czasie działania programu, a nie kompilacji, a jej rozmiary mogą być nieznane do momentu rozpoczęcia samego procesu przydzielania. Dla projektanta jest ona użyteczna w momencie, gdy zmienia się ilość pamięci, którą potrzebuje tworzony program lub też zmienny jest czas, w ciągu którego będzie ona używana, a dodatkowo wielkości te nie są znane przed uruchomieniem aplikacji. Na przykład, można zaimplementować przechowywanie w pamięci zawartości jakiegoś pliku lub danych wczytywanych

[3] Nazwa to relikt historii — jest to skrót od słów: *blok rozpoczęty od symbolu* (ang. *block started by symbol*).

z klawiatury. Ponieważ wielkość takiego pliku jest nieznana, a użytkownik może wprowadzić dowolną liczbę znaków z klawiatury, rozmiar bufora musi być zmienny, by programista dynamicznie go zwiększał, gdy danych zacznie przybywać.

Żadne zmienne języka C nie są zapisywane w pamięci dynamicznej. Na przykład, język C nie udostępnia mechanizmu, który pozwala na odczytanie struktury `pirate_ship` znajdującej się w takiej pamięci. Zamiast tego istnieje metoda pozwalająca na przydzielenie takiej ilości pamięci dynamicznej, która wystarczy, aby przechować w niej strukturę `pirate_ship`. Programista następnie używa tej pamięci poprzez posługiwanie się wskaźnikiem do niej — w tym przypadku, stosując wskaźnik `struct pirate_ship *`.

Klasycznym interfejsem języka C, pozwalającym na otrzymanie pamięci dynamicznej, jest funkcja `malloc()`:

```
#include <stdlib.h>

void * malloc (size_t size);
```

Poprawne jej wywołanie przydziela obszar pamięci, którego wielkość (w bajtach) określona jest w parametrze `size`. Funkcja zwraca wskaźnik do początku nowo przydzielonego regionu. Zawartość pamięci jest niezdefiniowana i nie należy oczekiwać, że będzie zawierać same zera. W przypadku błędu, funkcja `malloc()` zwraca `NULL` oraz ustawia zmienną `errno` na `ENOMEM`.

Użycie funkcji `malloc()` jest raczej proste, tak jak w przypadku poniższego przykładu przydzielającego określoną liczbę bajtów:

```
char *p;

/* przydziel 2 kB! */
p = malloc (2048);
if (!p)
   perror ("malloc");
```

Nieskomplikowany jest również kolejny przykład, przydzielający pamięć dla struktury:

```
struct treasure_map *map;

/*
* przydziel wystarczająco dużo pamięci, aby przechować strukturę treasure_map,
* a następnie przypisz adres tego obszaru do wskaźnika 'map'
*/
map = malloc (sizeof (struct treasure_map));
if (!map)
   perror ("malloc");
```

Język C automatycznie rzutuje wskaźniki typu `void` na dowolny typ, występujący podczas operacji przypisania. Dlatego też w przypadku powyższych przykładów, nie jest konieczne rzutowanie typu zwracanej wartości funkcji `malloc()` na typ l-wartości, używanej podczas operacji przypisania. Język programowania C++ nie wykonuje jednak automatycznego rzutowania wskaźnika `void`. Zgodnie z tym użytkownicy języka C++ muszą rzutować wyniki wywołania funkcji `malloc()`, tak jak pokazano to w poniższym przykładzie:

```
char *name;

/* przydziel 512 bajtów */
name = (char *) malloc (512);
if (!name)
   perror ("malloc");
```

Niektórzy programiści języka C preferują wykonywanie rzutowania wyników dowolnej funkcji, która zwraca wskaźnik `void`. Dotyczy to również funkcji `malloc()`. Ten styl programowania jest jednak niepewny z dwóch powodów. Po pierwsze, może on spowodować pominięcie błędu w przypadku, gdy wartość zwracana z funkcji kiedykolwiek ulegnie zmianie i nie będzie równa wskaźnikowi `void`. Po drugie, takie rzutowanie ukrywa błędy również w przypadku, gdy funkcja jest niewłaściwie zadeklarowana[4]. Pierwszy z tych powodów nie jest przyczyną powstawania problemów podczas użycia funkcji `malloc()`, natomiast drugi może już ich przysparzać.

Ponieważ funkcja `malloc()` może zwrócić wartość `NULL`, dlatego też jest szczególnie ważne, aby projektanci oprogramowania *zawsze* sprawdzali i obsługiwali przypadki błędów. W wielu programach funkcja `malloc()` nie jest używana bezpośrednio, lecz istnieje dla niej stworzony interfejs programowy (*wrapper*), który wyprowadza komunikat błędu i przerywa działanie programu, gdy zwraca ona wartość `NULL`. Zgodnie z konwencja nazewniczą, ten ogólny interfejs programowy zwany jest przez projektantów `xmalloc()`:

```
/* działa jak malloc(), lecz kończy wykonywanie programu w przypadku niepowodzenia */
void * xmalloc (size_t size)
{
   void *p;

   p = malloc (size);
   if (!p)
   {
      perror ("xmalloc");
      exit (EXIT_FAILURE);
   }

   return p;
}
```

Przydzielanie pamięci dla tablic

Dynamiczne przydzielanie pamięci może być skomplikowane, jeśli rozmiar danych, przekazany w parametrze `size`, jest również zmienny. Jednym z tego typu przykładów jest dynamiczne przydzielanie pamięci dla tablic, których rozmiar jednego elementu może być stały, lecz liczba alokowanych elementów jest zmienna.

Aby uprościć wykonywanie tej czynności, język C udostępnia funkcję `calloc()`:

```
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

Poprawne wywołanie funkcji `calloc()` zwraca wskaźnik do bloku pamięci o wielkości wystarczającej do przechowania tablicy o liczbie elementów określonej w parametrze `nr`. Każdy z elementów posiada rozmiar `size`. Zgodnie z tym ilość pamięci, przydzielona w przypadku użycia zarówno funkcji `malloc()`, jak i `calloc()`, jest taka sama (obie te funkcje mogą zwrócić więcej pamięci, niż jest to wymagane, lecz nigdy mniej):

```
int *x, *y;

x = malloc (50 * sizeof (int));
```

[4] Funkcje niezadeklarowane zwracają domyślnie wartości o typie `int`. Rzutowanie liczby całkowitej na wskaźnik nie jest wykonywane automatycznie i powoduje powstanie ostrzeżenia podczas kompilacji programu. Użycie rzutowania typów nie pozwala na generowanie takiego ostrzeżenia.

```
if (!x)
{
   perror ("malloc");
   return -1;
}

y = calloc (50, sizeof (int));
if (!y)
{
   perror ("calloc");
   return -1;
}
```

Zachowanie powyższych dwóch funkcji nie jest jednak identyczne. W przeciwieństwie do funkcji `malloc()`, która nie zapewnia, jaka będzie zawartość przydzielonej pamięci, funkcja `calloc()` zeruje wszystkie bajty w zwróconym obszarze pamięci. Dlatego też każdy z 50 elementów w tablicy liczb całkowitych `y` posiada wartość `0`, natomiast wartości elementów tablicy `x` są niezdefiniowane. Dopóki w programie nie ma potrzeby natychmiastowego zainicjalizowania wszystkich 50 wartości, programiści powinni używać funkcji `calloc()`, aby zapewnić, że elementy tablicy nie są wypełnione przypadkowymi danymi. Należy zauważyć, że zero binarne może być różne od zera występującego w liczbie zmiennoprzecinkowej!

Użytkownicy często chcą „wyzerować" pamięć dynamiczną, nawet wówczas, gdy nie używają tablic. W dalszej części tego rozdziału poddana analizie zostanie funkcja `memset()`, która dostarcza interfejsu pozwalającego na ustawienie wartości dla dowolnego bajta w obszarze pamięci. Funkcja `calloc()` wykonuje jednak tę operację szybciej, gdyż jądro może od razu udostępnić obszar pamięci, który wypełniony jest już zerami.

W przypadku błędu funkcja `calloc()`, podobnie jak `malloc()`, zwraca `−1` oraz ustawia zmienną `errno` na wartość `ENOMEM`.

Dlaczego w standardach nie zdefiniowano nigdy funkcji „przydziel i wyzeruj", różnej od `calloc()`, pozostaje tajemnicą. Projektanci mogą jednak w prosty sposób zdefiniować swój własny interfejs:

```
/* działa tak samo jak funkcja malloc(), lecz przydzielona pamięć zostaje wypełniona zerami */
void * malloc0 (size_t size)
{
   return calloc (1, size);
}
```

Można bez kłopotu połączyć funkcję `malloc0()` z poprzednio przedstawioną funkcją `xmalloc()`:

```
/* działa podobnie jak malloc(), lecz wypełnia pamięć zerami i przerywa działanie programu w przypadku błędu */
void * xmalloc0 (size_t size)
{
   void *p;

   p = calloc (1, size);
   if (!p)
   {
      perror ("xmalloc0");
      exit (EXIT_FAILURE);
   }

   return p;
}
```

Zmiana wielkości obszaru przydzielonej pamięci

Język C dostarcza interfejsu pozwalającego na zmianę wielkości (zmniejszenie lub powiększenie) istniejącego obszaru przydzielonej pamięci:

```
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

Poprawne wywołanie funkcji `realloc()` zmienia rozmiar regionu pamięci, wskazywanego przez `ptr`, na nową wartość, której wielkość podana jest w parametrze `size` i wyrażona w bajtach. Funkcja zwraca wskaźnik do obszaru pamięci posiadającego nowy rozmiar. Wskaźnik ten nie musi być równy wartości parametru `ptr`, który był używany w funkcji podczas wykonywania operacji powiększenia rozmiaru obszaru. Jeśli funkcja `realloc()` nie potrafi powiększyć istniejącego obszaru pamięci poprzez zmianę rozmiaru dla wcześniej przydzielonego miejsca, wówczas może ona zarezerwować pamięć dla nowego regionu pamięci o rozmiarze `size`, wyrażonym w bajtach, skopiować zawartość poprzedniego regionu w nowe miejsce, a następnie zwolnić niepotrzebny już obszar źródłowy. W przypadku każdej operacji zachowana zostaje zawartość dla takiej wielkości obszaru pamięci, która równa jest mniejszej wartości z dwóch rozmiarów: poprzedniego i aktualnego. Z powodu ewentualnego istnienia operacji kopiowania, wywołanie funkcji `realloc()`, które wykonuje powiększenie obszaru pamięci, może być stosunkowo kosztowne.

Jeśli `size` wynosi zero, rezultat jest taki sam jak w przypadku wywołania funkcji `free()` z parametrem `ptr`.

Jeśli parametr `ptr` jest równy `NULL`, wówczas rezultat wykonania operacji jest taki sam jak dla oryginalnej funkcji `malloc()`. Jeśli wskaźnik `ptr` jest różny od `NULL`, powinien zostać zwrócony przez wcześniejsze wykonanie jednej z funkcji `malloc()`, `calloc()` lub `realloc()`.

W przypadku błędu, funkcja `realloc()` zwraca `NULL` oraz ustawia zmienną `errno` na wartość `ENOMEM`. Stan obszaru pamięci, wskazywanego przez parametr `ptr`, pozostaje niezmieniony.

Rozważmy przykład programu, który zmniejsza obszar pamięci. Najpierw należy użyć funkcji `calloc()`, która przydzieli wystarczającą ilość pamięci, aby zapamiętać w niej dwuelementową tablicę struktur `map`:

```
struct map *p;

/* przydziel pamięć na dwie struktury 'map' */
p = calloc (2, sizeof (struct map));
if (!p)
{
   perror ("calloc");
   return -1;
}

/* w tym momencie można używać p[0] i p[1] ... */
```

Załóżmy, że jeden ze skarbów został już znaleziony, dlatego też nie ma potrzeby użycia drugiej mapy. Podjęto decyzję, że rozmiar obszaru pamięci zostanie zmieniony, a połowa przydzielonego wcześniej regionu zostanie zwrócona do systemu (operacja ta nie byłaby właściwie zbyt potrzebna, chyba że rozmiar struktury `map` byłby bardzo duży, a program rezerwowałby dla niej pamięć przez dłuższy czas):

```
struct map *r;

/* obecnie wymagana jest tylko pamięć dla jednej mapy */
r = realloc (p, sizeof (struct map));
if (!r)
{
   /* należy zauważyć, że 'p' jest wciąż poprawnym wskaźnikiem! */
   perror ("realloc");
   return -1;
}

/* tu można już używać wskaźnika 'r'... */

free (r);
```

W powyższym przykładzie, po wywołaniu funkcji `realloc()` zostaje zachowany element `p[0]`. Jakiekolwiek dane, które przedtem znajdowały się w tym elemencie, będą obecne również teraz. Jeśli wywołanie funkcji się nie powiedzie, należy zwrócić uwagę na to, że wskaźnik `p` nie zostanie zmieniony i stąd też będzie wciąż poprawny. Można go ciągle używać i w końcu należy go zwolnić. Jeśli wywołanie funkcji się powiedzie, należy zignorować wskaźnik `p` i zamiast niego użyć `r` (który jest przypuszczalnie równy `p`, gdyż najprawdopodobniej nastąpiła zmiana rozmiaru aktualnie przydzielonego obszaru). Obecnie programista odpowiedzialny będzie za zwolnienie pamięci dla wskaźnika `r`, gdy tylko przestanie on być potrzebny.

Zwalnianie pamięci dynamicznej

W przeciwieństwie do obszarów pamięci przydzielonych automatycznie, które same zostają zwolnione, gdy następuje przesunięcie wskaźnika stosu, dynamicznie przydzielone regiony pamięci pozostają trwałą częścią przestrzeni adresowej procesu, dopóki nie zostaną ręcznie zwolnione. Dlatego też programista odpowiedzialny jest za zwolnienie do systemu dynamicznie przydzielonej pamięci (oba rodzaje przydzielonej pamięci — statyczna i dynamiczna — zostają zwolnione, gdy cały proces kończy swoje działanie).

Pamięć, przydzielona za pomocą funkcji `malloc()`, `calloc()` lub `realloc()`, musi zostać zwolniona do systemu, jeśli nie jest już więcej używana. W tym celu stosuje się funkcję `free()`:

```
#include <stdlib.h>

void free (void *ptr);
```

Wywołanie funkcji `free()` zwalnia pamięć, wskazywaną przez wskaźnik `ptr`. Parametr `ptr` powinien być zainicjalizowany przez wartość zwróconą wcześniej przez funkcję `malloc()`, `calloc()` lub `realloc()`. Oznacza to, że nie można użyć funkcji `free()`, aby zwolnić fragment obszaru pamięci — na przykład połowę — poprzez przekazanie do niej parametru wskazującego na środek wcześniej przydzielonego obszaru.

Wskaźnik `ptr` może być równy `NULL`, co powoduje, że funkcja `free()` od razu wraca do procesu wywołującego. Dlatego też niepotrzebne jest sprawdzanie wskaźnika `ptr` przed wywołaniem funkcji `free()`.

Oto przykład użycia funkcji `free()`:

```
void print_chars (int n, char c)
{
   int i;

   for (i = 0; i < n; i++)
```

```
    {
        char *s;
        int j;

        /*
        * Przydziel i wyzeruj tablicę znaków o liczbie elementów równej i+2.
        * Należy zauważyć, że wywołanie 'sizeof (char)' zwraca zawsze wartość 1.
        */
        s = calloc (i + 2, 1);
        if (!s)
        {
            perror ("calloc");
            break;
        }

        for (j = 0; j < i + 1; j++)
            s[j] = c;

        printf ("%s\n", s);

        /* Wszystko zrobione, obecnie należy zwolnić pamięć. */
        free (s);
    }
}
```

Powyższy przykład przydziela pamięć dla `n` tablic typu `char`, zawierających coraz większą liczbę elementów, poczynając od dwóch (2 bajty), a kończąc na `n + 1` elementach (n + 1 bajtów). Wówczas dla każdej tablicy następuje w pętli zapisanie znaku `c` do poszczególnych jej elementów, za wyjątkiem ostatniego (pozostawiając tam bajt o wartości `0`, który jednocześnie jest ostatnim w danej tablicy), wyprowadzenie zawartości tablicy w postaci łańcucha znaków, a następnie zwolnienie przydzielonej dynamicznie pamięci.

Wywołanie funkcji `print_chars()` z parametrami `n` równym `5`, a `c` równym `X`, wymusi uzyskanie następującego wyniku:

```
X
XX
XXX
XXXX
XXXXX
```

Istnieją oczywiście dużo efektywniejsze metody pozwalające na zaimplementowanie takiej funkcji. Ważne jest jednak, że pamięć można dynamicznie przydzielać i zwalniać, nawet wówczas, gdy rozmiar i liczba przydzielonych obszarów znana jest tylko w momencie działania programu.

Systemy uniksowe, takie jak SunOS i SCO, udostępniają własny wariant funkcji `free()`, zwany `cfree()`, który w zależności od systemu działa tak samo jak `free()` lub posiada trzy parametry i wówczas zachowuje się jak funkcja `calloc()`. Funkcja `free()` w systemie Linux może obsłużyć pamięć uzyskaną dzięki użyciu dowolnego mechanizmu, służącego do jej przydzielania i już omówionego. Funkcja `cfree()` nie powinna być używana, za wyjątkiem zapewnienia wstecznej kompatybilności. Wersja tej funkcji dla Linuksa jest identyczna z `free()`.

Należy zauważyć, że gdyby w powyższym przykładzie nie użyto funkcji `free()`, pojawiłyby się pewne następstwa tego. Program mógłby nigdy nie zwolnić zajętego obszaru do systemu i co gorsze, stracić swoje jedyne odwołanie do pamięci — wskaźnik `s` — i przez to spowodować, że dostęp do niej stałby się w ogóle niemożliwy. Ten rodzaj błędu programistycznego

zwany jest *wyciekaniem pamięci* (ang. *memory leak*). Wyciekanie pamięci i tym podobne pomyłki, związane z pamięcią dynamiczną, są najczęstszymi i niestety najbardziej szkodliwymi błędami występującymi podczas programowania w języku C. Ponieważ język C zrzuca całą odpowiedzialność za zarządzanie pamięcią na programistów, muszą oni zwracać szczególną uwagę na wszystkie przydzielone obszary.

Równie często spotykaną pułapką języka C jest *używanie zasobów po ich zwolnieniu*. Problem ten występuje w momencie, gdy blok pamięci zostaje zwolniony, a następnie ponownie użyty. Gdy tylko funkcja `free()` zwolni dany obszar pamięci, program nie może już ponownie używać jego zawartości. Programiści powinni zwracać szczególną uwagę na zawieszone wskaźniki lub wskaźniki różne od `NULL`, które pomimo tego wskazują na niepoprawne obszary pamięci. Istnieją dwa powszechnie używane narzędzia pomagające w tych sytuacjach; są to *Electric Fence* i *valgrind*[5].

Wyrównanie

Wyrównanie danych dotyczy relacji pomiędzy ich adresem oraz obszarami pamięci udostępnianymi przez sprzęt. Zmienna posiadająca adres w pamięci, który jest wielokrotnością jej rozmiaru, zwana jest *zmienną naturalnie wyrównaną*. Na przykład, zmienna 32-bitowa jest naturalnie wyrównana, jeśli posiada adres w pamięci, który jest wielokrotnością 4 — oznacza to, że najniższe dwa bity adresu są równe zeru. Dlatego też typ danych, którego rozmiar wynosi 2^n bajtów, musi posiadać adres, którego *n* najmniej znaczących bitów jest ustawionych na zero.

Reguły, które dotyczą wyrównania, pochodzą od sprzętu. Niektóre architektury maszynowe posiadają bardzo rygorystyczne wymagania dotyczące wyrównania danych. W przypadku pewnych systemów, załadowanie danych, które nie są wyrównane, powoduje wygenerowanie pułapki procesora. Dla innych systemów dostęp do niewyrównanych danych jest bezpieczny, lecz związany z pogorszeniem sprawności działania. Podczas tworzenia kodu przenośnego należy unikać problemów związanych z wyrównaniem. Także wszystkie używane typy danych powinny być naturalnie wyrównane.

Przydzielanie pamięci wyrównanej

W większości przypadków kompilator oraz biblioteka języka C w sposób przezroczysty obsługują zagadnienia, związane z wyrównaniem. POSIX definiuje, że obszar pamięci, zwracany w wyniku wykonania funkcji `malloc()`, `calloc()` oraz `realloc()`, musi być prawidłowo wyrównany dla każdego standardowego typu danych języka C. W przypadku Linuksa funkcje te zawsze zwracają obszar pamięci, która wyrównana jest do adresu będącego wielokrotnością ośmiu bajtów w przypadku systemów 32-bitowych oraz do adresu, będącego wielokrotnością szesnastu bajtów dla systemów 64-bitowych.

Czasami programiści żądają przydzielenia takiego obszaru pamięci dynamicznej, który wyrównany jest do większego rozmiaru, posiadającego na przykład wielkość strony. Mimo istnienia różnych argumentacji, najbardziej podstawowym wymaganiem jest zdefiniowanie prawidłowo wyrównanych buforów, używanych podczas bezpośrednich operacji blokowych wejścia i wyjścia lub innej komunikacji między oprogramowaniem a sprzętem. W tym celu POSIX 1003.1d udostępnia funkcję zwaną `posix_memalign()`:

[5] Znajdują się one odpowiednio w następujących miejscach: *http://perens.com/FreeSoftware/ElectricFence/ oraz http://valgrind.org*.

```
/* należy użyć jednej z dwóch poniższych definicji - każda z nich jest odpowiednia */
#define _XOPEN_SOURCE 600
#define _GNU_SOURCE

#include <stdlib.h>
int posix_memalign (void **memptr, size_t alignment, size_t size);
```

Poprawne wywołanie funkcji `posix_memalign()` przydziela pamięć dynamiczną o rozmiarze przekazanym w parametrze `size` i wyrażonym w bajtach, zapewniając jednocześnie, że obszar ten zostanie wyrównany do adresu pamięci, będącego wielokrotnością parametru `alignment`. Parametr `alignment` musi być potęgą liczby 2 oraz wielokrotnością rozmiaru wskaźnika `void`. Adres przydzielonej pamięci zostaje umieszczony w parametrze `memptr`, a funkcja zwraca zero.

W przypadku błędu nie następuje przydzielenie pamięci, parametr `memptr` ma wartość nieokreśloną, a funkcja zwraca jedną z poniższych wartości kodów błędu:

`EINVAL`
: Parametr `alignment` nie jest potęgą liczby 2 lub wielokrotnością rozmiaru wskaźnika `void`.

`ENOMEM`
: Nie ma wystarczającej ilości pamięci, aby dokończyć rozpoczętą operację przydzielania pamięci.

Należy zauważyć, że zmienna `errno` nie zostaje ustawiona — funkcja bezpośrednio zwraca kod błędu.

Obszar pamięci, uzyskany za pomocą funkcji `posix_memalign()`, może zostać zwolniony przy użyciu `free()`. Sposób użycia funkcji jest prosty:

```
char *buf;
int ret;

/* przydziel 1 kB pamięci wyrównanej do adresu równego wielokrotności 256 bajtów */
ret = posix_memalign (&buf, 256, 1024);
if (ret)
{
   fprintf (stderr, "posix_memalign: %s\n", strerror (ret));
   return -1;
}

/ * tu można używać pamięci, wskazywanej przez 'buf'... */

free (buf);
```

Starsze interfejsy. Zanim w standardzie POSIX została zdefiniowana funkcja `posix_memalign()`, systemy BSD oraz SunOS udostępniały odpowiednio następujące interfejsy:

```
#include <malloc.h>

void * valloc (size_t size);
void * memalign (size_t boundary, size_t size);
```

Funkcja `valloc()` działa identycznie jak `malloc()`, za wyjątkiem tego, że przydzielona pamięć jest wyrównana do rozmiaru strony. Jak napisano w rozdziale 4., rozmiar systemowej strony można łatwo uzyskać po wywołaniu funkcji `getpagesize()`.

Funkcja `memalign()` jest podobna, lecz wyrównuje przydzieloną pamięć do rozmiaru przekazanego w parametrze `boundary` i wyrażonego w bajtach. Rozmiar ten musi być potęgą liczby 2. W poniższym przykładzie obie wspomniane funkcje alokacyjne zwracają blok pamięci o wielkości wystarczającej do przechowania struktury `ship`. Jest on wyrównany do rozmiaru strony:

```
struct ship *pirate, *hms;

pirate = valloc (sizeof (struct ship));
if (!pirate)
{
   perror ("valloc");
   return -1;
}

hms = memalign (getpagesize ( ), sizeof (struct ship));
if (!hms)
{
   perror ("memalign");
   free (pirate);
   return -1;
}

/* tu można używać obszaru pamięci wskazywanego przez 'pirate' i 'hms'... */

free (hms);
free (pirate);
```

W przypadku systemu Linux obszar pamięci, otrzymany za pomocą tych dwóch funkcji, może zostać zwolniony po wywołaniu funkcji `free()`. Nie musi tak być jednak w przypadku innych systemów uniksowych, gdyż niektóre z nich nie dostarczają żadnego mechanizmu pozwalającego na bezpieczne zwolnienie pamięci przydzielonej za pomocą wyżej wspomnianych funkcji. Dla programów, które powinny być przenośne, może nie istnieć inny wybór poza niezwalnianiem pamięci przydzielonej za pomocą tych interfejsów!

Programiści Linuksa powinni używać powyższych funkcji tylko wtedy, gdy należy zachować kompatybilność ze starszymi systemami; funkcja `posix_memalign()` jest lepsza. Użycie trzech wspomnianych funkcji jest niezbędne jedynie wtedy, gdy wymagany jest inny rodzaj wyrównania, niż dostarczony razem z funkcją `malloc()`.

Inne zagadnienia związane z wyrównaniem

Problemy związane z wyrównaniem obejmują większy obszar zagadnień niż tylko wyrównanie naturalne dla standardowych typów danych oraz dynamiczny przydział pamięci. Na przykład, typy niestandardowe oraz złożone posiadają bardziej skomplikowane wymagania niż typy standardowe. Ponadto, zagadnienia związane z wyrównaniem są szczególnie ważne w przypadku przypisywania wartości między wskaźnikami różnych typów oraz użycia rzutowania.

Typy niestandardowe. Niestandardowe i złożone typy danych posiadają większe wymagania dotyczące wyrównania przydzielonego obszaru pamięci. Zachowanie zwykłego wyrównania naturalnego nie jest wystarczające. W tych przypadkach stosuje się cztery poniższe reguły:

- Wyrównanie dla struktury jest równe wyrównaniu dla największego pod względem rozmiaru typu danych, z których zbudowane są jej pola. Na przykład, jeśli największy typ danych w strukturze jest 32-bitową liczbą całkowitą, która jest wyrównana do adresu będącego wielokrotnością czterech bajtów, wówczas sama struktura musi być także wyrównana do adresu będącego wielokrotnością co najmniej czterech bajtów.
- Użycie struktur wprowadza także konieczność stosowania wypełnienia, które jest wykorzystywane w celu zapewnienia, że każdy typ składowy będzie poprawnie wyrównany, zgodnie z jego wymaganiami. Dlatego też, jeśli po polu posiadającym typ `char` (o wyrównaniu prawdopodobnie równym jednemu bajtowi) pojawi się pole z typem `int` (posiadające

wyrównanie prawdopodobnie równe czterem bajtom), wówczas kompilator wstawi dodatkowe trzy bajty wypełnienia pomiędzy tymi dwoma polami o różnych typach danych, aby zapewnić, że `int` znajdzie się w obszarze wyrównanym do wielokrotności czterech bajtów. Programiści czasami porządkują pola w strukturze — na przykład, według malejącego rozmiaru typów składowych — aby zminimalizować obszar pamięci „tracony" na wypełnienie. Opcja kompilatora GCC, zwana `-Wpadded`, może pomóc w tym przypadku, ponieważ generuje ostrzeżenie w momencie, gdy kompilator wstawia domyślne wypełnienia.

- Wyrównanie dla unii jest równe wyrównaniu dla największego pod względem rozmiaru typu danych, z których zbudowane są jej pola.
- Wyrównanie dla tablicy jest równe wyrównaniu dla jej podstawowego typu danych. Dlatego też wymagania dla tablic są równe wymaganiu dotyczącemu pojedynczego elementu, z których się składają tablice. Zachowanie to powoduje, że wszystkie elementy tablicy posiadają wyrównanie naturalne.

Działania na wskaźnikach. Ponieważ kompilator w sposób przezroczysty obsługuje większość żądań związanych z wyrównaniem, dlatego też, aby doświadczyć ewentualnych problemów, wymagany jest większy wysiłek. Mimo to jest nieprawdą, że nie istnieją komplikacje związane z wyrównaniem, gdy używa się wskaźników i rzutowania.

Dostęp do danych poprzez rzutowanie wskaźnika z bloku pamięci o mniejszej wartości wyrównania na blok, posiadający większą wartość wyrównania, może spowodować, że dane te nie będą właściwie wyrównane dla typu o większym rozmiarze. Na przykład, przypisanie zmiennej `c` do `badnews` w poniższym fragmencie kodu powoduje, że zmienna ta będzie zrzutowana na typ `unsigned long`:

```
char greeting[] = "Ahoj Matey";
char *c = greeting[1];
unsigned long badnews = *(unsigned long *) c;
```

Typ `unsigned long` jest najprawdopodobniej wyrównany do adresu będącego wielokrotnością ośmiu bajtów; zmienna `c` prawie na pewno przesunięta jest o 1 bajt poza tę granicę. Odczytanie zmiennej `c` podczas wykonywania rzutowania spowoduje powstanie błędu wyrównania. W zależności od architektury może być to przyczyną różnych zachowań, poczynając od mniej ważnych, np. pogorszenie sprawności działania, a kończąc na poważnych, jak załamanie programu. W architekturach maszynowych, które potrafią wykryć, lecz nie mogą poprawnie obsłużyć błędów wyrównania, jądro wysyła do takich niepoprawnych procesów sygnał `SIGBUS`, który przerywa ich działanie. Sygnały zostaną omówione w rozdziale 9.

Przykłady podobne do powyższego są częściej spotykane, niż sądzimy. Niepoprawne konstrukcje programowe, spotykane w świecie realnym, nie będą wyglądać tak bezmyślnie, lecz będą najprawdopodobniej trudniejsze do wykrycia.

Zarządzanie segmentem danych

Od zawsze system Unix udostępniał interfejsy pozwalające na bezpośrednie zarządzanie segmentem danych. Jednak większość programów nie posiada bezpośredniego dostępu do tych interfejsów, ponieważ funkcja `malloc()` i inne sposoby przydzielania pamięci są łatwiejsze w użyciu, a jednocześnie posiadają większe możliwości. Interfejsy te zostaną jednak omówione,

aby zaspokoić ciekawość czytelników i udostępnić dociekliwym programistom metodę pozwalającą na zaimplementowanie swojego własnego mechanizmu przydzielania pamięci, opartego na stercie:

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t increment);
```

Funkcje te dziedziczą swoje nazwy z dawnych systemów uniksowych, dla których sterta i stos znajdowały się w tym samym segmencie. Przydzielanie obszarów pamięci dynamicznej na stercie powoduje jej narastanie od dolnej części segmentu, w kierunku adresów wyższych; stos rośnie w kierunku przeciwnym — od szczytu segmentu do niższych adresów. Linia graniczna pomiędzy tymi dwoma strukturami danych zwana jest *podziałem* lub *punktem podziału* (ang. *break* lub *break point*). W nowoczesnych systemach operacyjnych, w których segment danych posiada swoje własne odwzorowanie pamięci, końcowy adres tego odwzorowania w dalszym ciągu zwany jest punktem podziału.

Wywołanie funkcji `brk()` ustawia punkt podziału (koniec segmentu danych) na adres przekazany w parametrze `end`. W przypadku sukcesu, funkcja zwraca wartość `0`. W przypadku błędu, zwraca `−1` oraz ustawia zmienną `errno` na `ENOMEM`.

Wywołanie funkcji `sbrk()` zwiększa adres końca segmentu o wartość przekazaną w parametrze `increment`, który może być przyrostem dodatnim lub ujemnym. Funkcja `sbrk()` zwraca uaktualnioną wartość położenia punktu podziału. Dlatego też użycie parametru `increment` równego zeru powoduje wyprowadzenie aktualnej wartości położenia punktu podziału:

```
printf ("Aktualny punkt podziału posiada adres %p\n", sbrk (0));
```

Oba standardy — POSIX i C — celowo nie definiują żadnej z powyższych funkcji. Prawie wszystkie systemy uniksowe wspierają jednak jedną lub obie te funkcje. Programy przenośne powinny używać interfejsów zdefiniowanych w standardach.

Anonimowe odwzorowania w pamięci

W celu wykonania operacji przydzielania pamięci, zaimplementowanej w bibliotece *glibc*, używany jest segment danych oraz odwzorowania pamięci. Klasyczną metodą, zastosowaną w celu implementacji funkcji `malloc()`, jest podział segmentu danych na ciąg partycji o rozmiarach potęgi liczby 2 oraz zwracanie tego obszaru, który najlepiej pasuje do żądanej wielkości. Zwalnianie pamięci jest prostym oznaczaniem, że dana partycja jest „wolna". Kiedy graniczące ze sobą partycje są nieużywane, mogą zostać połączone w jeden większy obszar pamięci. Jeśli szczyt sterty jest zupełnie nieprzydzielony, system może użyć funkcji `brk()`, aby obniżyć adres położenia punktu podziału, a przez to zmniejszyć rozmiar tej struktury danych i zwrócić pamięć do jądra.

Algorytm ten zwany jest *schematem przydziału wspieranej pamięci* (ang. *buddy memory allocation scheme*). Posiada takie zalety jak prędkość i prostota, ale również wady w postaci dwóch rodzajów fragmentacji. *Fragmentacja wewnętrzna* (ang. *internal fragmentation*) występuje wówczas, gdy więcej pamięci, niż zażądano, zostanie użyte w celu wykonania operacji przydziału. Wynikiem tego jest nieefektywne użycie dostępnej pamięci. *Fragmentacja zewnętrzna* (ang. *external fragmentation*) występuje wówczas, gdy istnieje wystarczająca ilość pamięci, aby zapewnić wykonanie operacji przydziału, lecz jest ona podzielona na dwa lub więcej niesąsiadujących ze sobą frag-

mentów. Fragmentacja ta może powodować nieefektywne użycie pamięci (ponieważ może zostać użyty większy, mniej pasujący blok) lub niepoprawne wykonanie operacji jej przydziału (jeśli nie ma innych bloków).

Ponadto, schemat ten pozwala, aby pewien przydzielony obszar mógł „unieruchomić" inny, co może spowodować, że biblioteka *glibc* nie będzie mogła zwrócić zwolnionej pamięci do jądra. Załóżmy, że istnieją dwa przydzielone obszary pamięci: blok *A* i blok *B*. Blok *A* znajduje się dokładnie w punkcie podziału, a blok *B* zaraz pod nim. Nawet jeśli program zwolni blok *B*, biblioteka *glibc* nie będzie mogła uaktualnić położenia punktu podziału, dopóki blok *A* również nie zostanie zwolniony. W ten sposób aplikacje, których czas życia w systemie jest długi, mogą unieruchomić wszystkie inne przydzielone obszary pamięci.

Nie zawsze jest to problemem, gdyż biblioteka *glibc* nie zwraca w sposób rutynowy pamięci do systemu[6]. Sterta zazwyczaj nie zostaje zmniejszona po każdej operacji zwolnienia pamięci. Zamiast tego biblioteka *glibc* zachowuje zwolnioną pamięć, aby użyć jej w następnej operacji przydzielania. Tylko wówczas, gdy rozmiar sterty jest znacząco większy od ilości przydzielonej pamięci, biblioteka *glibc* faktycznie zmniejsza wielkość segmentu danych. Przydział dużej ilości pamięci może jednak przeszkodzić temu zmniejszeniu.

Zgodnie z tym, w przypadku przydziałów dużej ilości pamięci, w bibliotece *glibc* nie jest używana sterta. Biblioteka *glibc* tworzy *anonimowe odwzorowanie w pamięci*, aby zapewnić poprawne wykonanie żądania przydziału. Anonimowe odwzorowania w pamięci są podobne do odwzorowań dotyczących plików i omówionych w rozdziale 4., za wyjątkiem tego, że nie są związane z żadnym plikiem — stąd też przydomek „anonimowy". Takie anonimowe odwzorowanie jest po prostu dużym blokiem pamięci, wypełnionym zerami i gotowym do użycia. Należy traktować go jako nową stertę używaną wyłącznie w jednej operacji przydzielania pamięci. Ponieważ takie odwzorowania są umieszczane poza stertą, nie przyczyniają się do fragmentacji segmentu danych.

Przydzielanie pamięci za pomocą anonimowych odwzorowań ma kilka zalet:

- Nie występuje fragmentacja. Gdy program nie potrzebuje już anonimowego odwzorowania w pamięci, jest ono usuwane, a pamięć zostaje natychmiast zwrócona do systemu.
- Można zmieniać rozmiar anonimowych odwzorowań w pamięci, posiadają one modyfikowane uprawnienia, a także mogą otrzymywać poradę — podobnie, jak ma to miejsce w przypadku zwykłych odwzorowań (szczegóły w rozdziale 4.).
- Każdy przydział pamięci realizowany jest w oddzielnym odwzorowaniu. Nie ma potrzeby użycia globalnej sterty.

Istnieją również wady używania anonimowych odwzorowań w pamięci, w porównaniu z użyciem sterty:

- Rozmiar każdego odwzorowania w pamięci jest całkowitą wielokrotnością rozmiaru strony systemowej. Zatem takie operacje przydziałów, dla których rozmiary nie są całkowitą wielokrotnością rozmiaru strony, generują powstawanie nieużywanych obszarów „wolnych". Problem przestrzeni wolnej dotyczy głównie małych obszarów przydziału, dla których pamięć nieużywana jest stosunkowo duża w porównaniu z rozmiarem przydzielonego bloku.

[6] W celu przydzielania pamięci, biblioteka *glibc* używa również dużo bardziej zaawansowanego algorytmu niż zwykłego schematu przydziału wspieranej pamięci. Algorytm ten zwany jest *algorytmem areny* (ang. *arena algorithm*).

- Tworzenie nowego odwzorowania w pamięci wymaga większego nakładu pracy niż zwracanie pamięci ze sterty, które może w ogóle nie obciążać jądra. Im obszar przydziału jest mniejszy, tym to zjawisko jest bardziej widoczne.

  Porównując zalety i wady, można stwierdzić, że funkcja `malloc()` w bibliotece *glibc* używa segmentu danych, aby zapewnić poprawne wykonanie operacji przydziału niewielkich obszarów, natomiast anonimowych odwzorowań w pamięci, aby zapewnić przydzielenie dużych obszarów. Próg działania jest konfigurowalny (szczegóły w podrozdziale Zaawansowane operacje przydziału pamięci, znajdującym się w dalszej części tego rozdziału) i może być inny dla każdej wersji biblioteki *glibc*. Obecnie próg wynosi 128 kB: operacje przydziału o obszarach mniejszych lub równych 128 kB używają sterty, natomiast większe przydziały korzystają z anonimowych odwzorowań w pamięci.

Tworzenie anonimowych odwzorowań w pamięci

Wymuszenie użycia mechanizmu odwzorowania w pamięci zamiast wykorzystania sterty w celu wykonania określonego przydziału, kreowanie własnego systemu zarządzającego przydziałem pamięci, ręczne tworzenie anonimowego odwzorowania w pamięci — te wszystkie operacje są łatwe do zrealizowania w systemie Linux. W rozdziale 4. napisano, że odwzorowanie w pamięci może zostać utworzone przez funkcję systemową `mmap()`, natomiast usunięte przez funkcję systemową `munmap()`:

```
#include <sys/mman.h>

void * mmap (void *start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap (void *start, size_t length);
```

Kreowanie anonimowego odwzorowania w pamięci jest nawet prostsze niż tworzenie odwzorowania opartego na pliku, ponieważ nie trzeba tego pliku otwierać i nim zarządzać. Podstawową różnicą między tymi dwoma rodzajami odwzorowania jest specjalny znacznik, wskazujący, że dane odwzorowanie jest anonimowe.

Oto przykład:

```
void *p;

p = mmap (NULL,                          /* nieważne, w jakim miejscu pamięci */
  512 * 1024,                            /* 512 kB */
    PROT_READ | PROT_WRITE,              /* zapis/odczyt */
      MAP_ANONYMOUS | MAP_PRIVATE,       /* odwzorowanie anonimowe i prywatne */
        -1,                              /* deskryptor pliku (ignorowany) */
           0);                           /* przesunięcie (ignorowane) */

if (p == MAP_FAILED)
   perror ("mmap");
else
   /* 'p' wskazuje na obszar 512 kB anonimowej pamięci... */
```

W większości anonimowych odwzorowań parametry funkcji `mmap()` są takie same jak w powyższym przykładzie, oczywiście za wyjątkiem rozmiaru, przekazanego w parametrze `length` i wyrażonego w bajtach, który jest określany przez programistę. Pozostałe parametry są następujące:

- Pierwszy parametr, `start`, ustawiony jest na wartość `NULL`, co oznacza, że anonimowe odwzorowanie może rozpocząć się w dowolnym miejscu w pamięci — decyzja w tym przypadku należy do jądra. Podawanie wartości różnej od `NULL` jest dopuszczalne, dopóki

jest ona wyrównana do wielkości strony, lecz ogranicza to przenośność. Położenie odwzorowania jest rzadko wykorzystywane przez programy.

- Parametr `prot` zwykle ustawia oba bity `PROT_READ` oraz `PROT_WRITE`, co powoduje, że odwzorowanie posiada uprawienia do odczytu i zapisu. Odwzorowanie bez uprawnień nie ma sensu, gdyż nie można z niego czytać ani do niego zapisywać. Z drugiej strony, zezwolenie na wykonywanie kodu z anonimowego odwzorowania jest rzadko potrzebne, a jednocześnie tworzy potencjalną lukę bezpieczeństwa.
- Parametr `flags` ustawia bit `MAP_ANONYMOUS`, który oznacza, że odwzorowanie jest anonimowe, oraz bit `MAP_PRIVATE`, który nadaje odwzorowaniu status prywatności.
- Parametry `fd` i `offset` są ignorowane, gdy ustawiony jest znacznik `MAP_ANONYMOUS`. Niektóre starsze systemy oczekują jednak, że w parametrze `fd` zostanie przekazana wartość `-1`, dlatego też warto to uczynić, gdy ważnym czynnikiem jest przenośność.

Pamięć, otrzymana za pomocą mechanizmu anonimowego odwzorowania, wygląda tak samo jak pamięć ze sterty. Jedną korzyścią z użycia anonimowego odwzorowania jest to, że strony są już wypełnione zerami. Jest to wykonywane bez jakichkolwiek kosztów, ponieważ jądro odwzorowuje anonimowe strony aplikacji na stronę wypełnioną zerami, używając do tego celu mechanizmu kopiowania podczas zapisu. Dlatego też nie jest wymagane użycie funkcji `memset()` dla zwróconego obszaru pamięci. Faktycznie istnieje jedna korzyść z użycia funkcji `calloc()` zamiast zestawu `malloc()`oraz `memset()`: biblioteka *glibc* jest poinformowana, że obszar anonimowego odwzorowania jest już wypełniony zerami, a funkcja `calloc()`, po poprawnym przydzieleniu pamięci, nie wymaga jawnego jej zerowania.

Funkcja systemowa `munmap()` zwalnia anonimowe odwzorowanie, zwracając przydzieloną pamięć do jądra:

```
int ret;

/* wykonano wszystkie działania, związane z użyciem wskaźnika 'p', dlatego należy zwrócić 512 kB pamięci */
ret = munmap (p, 512 * 1024);
if (ret)
   perror ("munmap");
```

Szczegóły użycia funkcji `mmap()`, `munmap()` oraz ogólny opis mechanizmu odwzorowania znajdują się w rozdziale 4.

Odwzorowanie pliku /dev/zero

Inne systemy operacyjne, takie jak BSD, nie posiadają znacznika `MAP_ANONYMOUS`. Zamiast tego zaimplementowane jest dla nich podobne rozwiązanie, przy użyciu odwzorowania specjalnego pliku urządzenia */dev/zero*. Ten plik urządzenia dostarcza takiej samej semantyki jak anonimowa pamięć. Odwzorowanie zawiera strony uzyskane za pomocą mechanizmu kopiowania podczas zapisu, wypełnione zerami; dlatego też zachowanie to jest takie samo jak w przypadku anonimowej pamięci.

Linux zawsze posiadał urządzenie */dev/zero* oraz udostępniał możliwość odwzorowania tego pliku i uzyskania obszaru pamięci wypełnionego zerami. Rzeczywiście, zanim wprowadzono znacznik `MAP_ANONYMOUS`, programiści w Linuksie używali powyższego rozwiązania. Aby

zapewnić wsteczną kompatybilność ze starszymi wersjami Linuksa lub przenośność do innych systemów Uniksa, projektanci w dalszym ciągu mogą używać pliku urządzenia */dev/zero*, aby stworzyć anonimowe odwzorowanie. Operacja ta nie różni się od tworzenia odwzorowania dla innych plików:

```
void *p;
int fd;

/* otwórz plik /dev/zero do odczytu i zapisu */
fd = open ("/dev/zero", O_RDWR);
if (fd < 0)
{
   perror ("open");
   return -1;
}

/* odwzoruj obszar [0, rozmiar strony) dla urządzenia /dev/zero */
p = mmap (NULL,                    /* nieważne, w jakim miejscu pamięci */
  getpagesize ( ),                 /* odwzoruj jedną stronę */
    PROT_READ | PROT_WRITE,        /* uprawnienia odczytu i zapisu */
      MAP_PRIVATE,                 /* odwzorowanie prywatne */
        fd,                        /* odwzoruj plik /dev/zero */
          0);                      /* bez przesunięcia */
if (p == MAP_FAILED)
{
   perror ("mmap");
   if (close (fd))
      perror ("close");
   return -1;
}

/* zamknij plik /dev/zero, jeśli nie jest już potrzebny */
if (close (fd))
   perror ("close");

/* wskaźnik 'p' wskazuje na jedną stronę w pamięci, można go używać... */
```

Pamięć, otrzymana za pomocą powyżej przedstawionego sposobu, może oczywiście zostać zwolniona przy użyciu funkcji `munmap()`.

Ta metoda generuje dodatkowe obciążenie przez użycie funkcji systemowej, otwierającej i zamykającej plik urządzenia. Dlatego też wykorzystanie pamięci anonimowej jest rozwiązaniem szybszym.

Zaawansowane operacje przydziału pamięci

Wiele operacji przydziału pamięci, omówionych w tym rozdziale, jest ograniczanych i sterowanych przez parametry jądra, które mogą zostać modyfikowane przez programistę. Aby to wykonać, należy użyć funkcji `mallopt()`:

```
#include <malloc.h>

int mallopt (int param, int value);
```

Wywołanie funkcji `mallopt()` ustawia parametr związany z zarządzaniem pamięcią, którego nazwa przekazana jest w argumencie `param`. Parametr ten zostaje ustawiony na wartość równą argumentowi `value`. W przypadku sukcesu funkcja zwraca wartość niezerową; w przypadku błędu zwraca `0`. Należy zauważyć, że funkcja `mallopt()` nie ustawia zmiennej `errno`. Najczęściej

jej wywołanie również kończy się sukcesem, dlatego też nie należy optymistycznie podchodzić do zagadnienia uzyskiwania użytecznej informacji z jej kodu powrotu.

Linux wspiera obecnie sześć wartości dla parametru `param`, które zdefiniowane są w pliku nagłówkowym `<malloc.h>`:

`M_CHECK_ACTION`
: Wartość zmiennej środowiskowej `MALLOC_CHECK_` (omówiona w następnym podrozdziale).

`M_MMAP_MAX`
: Maksymalna liczba odwzorowań, które mogą zostać udostępnione przez system, aby poprawnie zrealizować żądania przydzielania pamięci dynamicznej. Gdy to ograniczenie zostanie osiągnięte, wówczas dla kolejnych przydziałów pamięci zostanie użyty segment danych, dopóki jedno z odwzorowań nie zostanie zwolnione. Wartość `0` całkowicie uniemożliwia użycie mechanizmu anonimowych odwzorowań jako podstawy do wykonywania operacji przydziału pamięci dynamicznej.

`M_MMAP_THRESHOLD`
: Wielkość progu (wyrażona w bajtach), powyżej którego żądanie przydziału pamięci zostanie zrealizowane za pomocą anonimowego odwzorowania zamiast udostępnienia segmentu danych. Należy zauważyć, że przydziały mniejsze od tego progu mogą również zostać zrealizowane za pomocą anonimowych odwzorowań, ze względu na swobodę postępowania pozostawioną systemowi. Wartość `0` umożliwia użycie anonimowych odwzorowań dla wszystkich operacji przydziału, stąd też w rzeczywistości nie zezwala na wykorzystanie dla nich segmentu danych.

`M_MXFAST`
: Maksymalny rozmiar (wyrażony w bajtach) podajnika szybkiego. *Podajniki szybkie* (ang. *fast bins*) są specjalnymi fragmentami pamięci na stercie, które nigdy nie zostają połączone z sąsiednimi obszarami i nie są zwrócone do systemu. Pozwala to na wykonywanie bardzo szybkich operacji przydziału, kosztem zwiększonej fragmentacji. Wartość `0` całkowicie uniemożliwia użycie podajników szybkich.

`M_TOP_PAD`
: Wartość uzupełnienia (w bajtach) użytego podczas zmiany rozmiaru segmentu danych. Gdy biblioteka *glibc* wykonuje funkcję `brk()`, aby zwiększyć rozmiar segmentu danych, może zażyczyć sobie więcej pamięci, niż w rzeczywistości potrzebuje, w nadziei na to, że dzięki temu w najbliższej przyszłości nie będzie konieczne wykonanie kolejnego wywołania tejże funkcji. Podobnie dzieje się w przypadku, gdy biblioteka *glibc* zmniejsza rozmiar segmentu danych — zachowuje ona dla siebie pewną ilość pamięci, zwracając do systemu mniej, niż mogłaby naprawdę oddać. Ten dodatkowy obszar pamięci jest omawianym *uzupełnieniem*. Wartość `0` uniemożliwia całkowicie użycie wypełnienia.

`M_TRIM_THRESHOLD`
: Minimalna ilość wolnej pamięci (w bajtach), która może istnieć na szczycie segmentu danych. Jeśli liczba ta będzie mniejsza od podanego progu, biblioteka *glibc* wywoła funkcję `brk()`, aby zwrócić pamięć do jądra.

Standard XPG, który w luźny sposób definiuje funkcję `mallopt()`, określa trzy inne parametry: `M_GRAIN`, `M_KEEP` oraz `M_NLBLKS`. Linux również je definiuje, lecz ustawianie dla nich wartości nie powoduje żadnych zmian. W tabeli 8.1. znajduje się pełny opis wszystkich poprawnych parametrów oraz odpowiednich dla nich domyślnych wartości. Podane są również zakresy akceptowalnych wartości.

Tabela 8.1. Parametry funkcji `mallopt()`

Parametr	Źródło pochodzenia	Wartość domyślna	Poprawne wartości	Wartości specjalne
`M_CHECK_ACTION`	Specyficzny dla Linuksa	`0`	`0 – 2`	
`M_GRAIN`	Standard XPG	Brak wsparcia w Linuksie	`>= 0`	
`M_KEEP`	Standard XPG	Brak wsparcia w Linuksie	`>= 0`	
`M_MMAP_MAX`	Specyficzny dla Linuksa	`64 * 1024`	`>= 0`	`0` uniemożliwia użycie `mmap()`
`M_MMAP_THRESHOLD`	Specyficzny dla Linuksa	`128 * 1024`	`>= 0`	`0` uniemożliwia użycie sterty
`M_MXFAST`	Standard XPG	`64`	`0 – 80`	`0` uniemożliwia użycie podajników szybkich
`M_NLBLKS`	Standard XPG	Brak wsparcia w Linuksie	`>= 0`	
`M_TOP_PAD`	Specyficzny dla Linuksa	`0`	`>= 0`	`0` uniemożliwia użycie uzupełnienia

Dowolne wywołanie funkcji `mallopt()` w programach musi wystąpić przed pierwszym użyciem funkcji `malloc()` lub innych interfejsów, służących do przydzielania pamięci. Użycie jest proste:

```
int ret;

/* użyj funkcji mmap( ) dla wszystkich przydziałów pamięci większych od 64 kB */
ret = mallopt (M_MMAP_THRESHOLD, 64 * 1024);
if (!ret)
   fprintf (stderr, "Wywołanie funkcji mallopt() nie powiodło się!\n");
```

Dokładne dostrajanie przy użyciu funkcji malloc_usable_size() oraz malloc_trim()

Linux dostarcza kilku funkcji, które pozwalają na niskopoziomową kontrolę działania systemu przydzielania pamięci dla biblioteki *glibc*. Pierwsza z tych funkcji pozwala na uzyskanie informacji, ile faktycznie dostępnych bajtów zawiera dany obszar przydzielonej pamięci:

```
#include <malloc.h>

size_t malloc_usable_size (void *ptr);
```

Poprawne wywołanie funkcji `malloc_usable_size()` zwraca rzeczywisty rozmiar przydziału dla obszaru pamięci wskazywanego przez `ptr`. Ponieważ biblioteka *glibc* może zaokrąglać wielkości przydziałów, aby dopasować się do istniejącego fragmentu pamięci, przydzielonego do anonimowego odwzorowania, dlatego też wielkość przestrzeni dla danego przydziału, nadającej się do użytku, może być większa od tej, jaką zażądano. Oczywiście obszary przydziałów pamięci nie będą nigdy mniejsze od tych, jakie są wymagane. Oto przykład użycia funkcji:

```
size_t len = 21;
size_t size;
char *buf;

buf = malloc (len);
if (!buf)
{
   perror ("malloc");
```

```
    return -1;
}

size = malloc_usable_size (buf);

/* w rzeczywistości można użyć 'size' bajtów z obszaru pamięci 'buf'... */
```

Wywołanie drugiej funkcji nakazuje bibliotece *glibc*, aby natychmiast zwróciła całą zwolnioną pamięć do jądra:

```
#include <malloc.h>

int malloc_trim (size_t padding);
```

Poprawne wywołanie funkcji `malloc_trim()` powoduje maksymalne zmniejszenie rozmiaru segmentu danych, za wyjątkiem obszarów uzupełnień, które są zarezerwowane. Następnie funkcja zwraca `1`. W przypadku błędu zwraca `0`. Zazwyczaj biblioteka *glibc* samodzielnie przeprowadza takie operacje zmniejszania rozmiaru segmentu danych, gdy tylko wielkość pamięci zwolnionej osiąga wartość `M_TRIM_THRESHOLD`. Biblioteka używa uzupełnienia określonego w parametrze `M_TOP_PAD`.

Programista nie będzie potrzebował nigdy użyć obu wspomnianych funkcji do niczego innego niż tylko celów edukacyjnych i wspomagających uruchamianie programów. Nie są one przenośne i udostępniają programowi użytkownika niskopoziomowe szczegóły systemu przydzielania pamięci, zaimplementowanego w bibliotece *glibc*.

Uruchamianie programów, używających systemu przydzielania pamięci

Programy mogą ustawiać zmienną środowiskową `MALLOC_CHECK_`, aby umożliwić poszerzone wspomaganie podczas uruchamiania programów wykorzystujących podsystem pamięci. Opcja poszerzonego wspomagania uruchamiania działa kosztem zmniejszenia efektywności operacji przydzielania pamięci, lecz obciążenie to jest często tego warte podczas tworzenia aplikacji i w trakcie jej uruchamiania.

Ponieważ zmienna środowiskowa steruje procesem wspomagania uruchamiania, dlatego też nie istnieje potrzeba, aby ponownie kompilować program. Na przykład, można wykonać proste polecenie, podobne do poniżej przedstawionego:

```
$ MALLOC_CHECK_=1 ./rudder
```

Jeśli zmienna `MALLOC_CHECK_` zostanie ustawiona na `0`, podsystem pamięci w sposób automatyczny zignoruje wszystkie błędy. W przypadku, gdy będzie ona równa `1`, na standardowe wyjście błędów `stderr` zostanie wysłany komunikat informacyjny. Jeśli zmienna ta będzie równa `2`, wykonanie programu zostanie natychmiast przerwane przy użyciu funkcji `abort()`. Ponieważ zmienna `MALLOC_CHECK_` modyfikuje zachowanie działającego programu, jest ignorowana przez aplikacje posiadające ustawiony bit SUID.

Otrzymywanie danych statystycznych

Linux dostarcza funkcji `mallinfo()`, która może zostać użyta w celu uzyskania danych statystycznych dotyczących działania systemu przydzielania pamięci:

```
#include <malloc.h>

struct mallinfo mallinfo (void);
```

Wywołanie funkcji `mallinfo()` zwraca dane statystyczne zapisane w strukturze `mallinfo`. Struktura zwracana jest przez wartość, a nie przez wskaźnik. Jej zawartość jest również zdefiniowana w pliku nagłówkowym `<malloc.h>`:

```
/* wszystkie rozmiary w bajtach */
struct mallinfo
{
   int arena;     /* rozmiar segmentu danych, używanego przez funkcję malloc */
   int ordblks;   /* liczba wolnych fragmentów pamięci */
   int smblks;    /* liczba podajników szybkich */
   int hblks;     /* liczba anonimowych odwzorowań */
   int hblkhd;    /* rozmiar anonimowych odwzorowań */
   int usmblks;   /* maksymalny rozmiar całkowitego przydzielonego obszaru */
   int fsmblks;   /* rozmiar dostępnych podajników szybkich */
   int uordblks;  /* rozmiar całkowitego przydzielonego obszaru */
   int fordblks;  /* rozmiar dostępnych fragmentów pamięci */
   int keepcost;  /* rozmiar obszaru, który może zostać zwrócony do systemu przy użyciu funkcji malloc_trim() */
};
```

Użycie funkcji jest proste:

```
struct mallinfo m;

m = mallinfo ( );

printf ("Liczba wolnych fragmentów pamięci: %d\n", m.ordblks);
```

Linux dostarcza również funkcji `malloc_stats()`, która wyprowadza na standardowe wyjście błędów dane statystyczne związane z podsystemem pamięci:

```
#include <malloc.h>

void malloc_stats (void);
```

Wywołanie funkcji `malloc_stats()` dla programu, który intensywnie używa pamięci, powoduje wyprowadzenie kilku większych liczb:

```
Arena 0:
system bytes     =  865939456
in use bytes     =  851988200
Total (incl. mmap):
system bytes     = 3216519168
in use bytes     = 3202567912
max mmap regions =      65536
max mmap bytes   = 2350579712
```

Przydziały pamięci wykorzystujące stos

Wszystkie mechanizmy omówione do tej pory, dotyczące wykonywania operacji przydziału pamięci dynamicznej, używały sterty lub odwzorowań w pamięci, aby zrealizować przydzielenie obszaru tejże pamięci. Należało tego oczekiwać, gdyż sterta i odwzorowania w pamięci są z definicji bardzo dynamicznymi strukturami. Inną, powszechnie używaną strukturą w przestrzeni adresowej programu jest stos, w którym zapamiętane są *automatyczne zmienne* dla aplikacji.

Nie istnieje jednak przeciwwskazanie, aby programista nie mógł używać stosu dla realizowania operacji przydzielania pamięci dynamicznej. Dopóki taka metoda przydziału pamięci nie przepełni stosu, może być prosta w realizacji i powinna działać zupełnie dobrze. Aby dynamicznie przydzielić pamięć na stosie, należy użyć funkcji systemowej `alloca()`:

```
#include <alloca.h>

void * alloca (size_t size);
```

W przypadku sukcesu, wywołanie funkcji `alloca()` zwraca wskaźnik do obszaru pamięci posiadającego rozmiar przekazany w parametrze `size` i wyrażony w bajtach. Pamięć ta znajduje się na stosie i zostaje automatycznie zwolniona, gdy wywołująca funkcja kończy swoje działanie. Niektóre implementacje zwracają wartość `NULL` w przypadku błędu, lecz dla większości z nich wywołanie funkcji `alloca()` nie może się nie udać lub nie jest możliwe informowanie o niepoprawnym jej wykonaniu. Na błąd wskazuje przepełniony stos.

Użycie jest identyczne jak w przypadku funkcji `malloc()`, lecz nie trzeba (w rzeczywistości *nie wolno*) zwalniać przydzielonej pamięci. Poniżej przedstawiony zostanie przykład funkcji, która otwiera dany plik z systemowego katalogu konfiguracyjnego (równego prawdopodobnie */etc*), lecz dla zwiększenia przenośności jego nazwa określana jest w czasie wykonania programu. Funkcja musi przydzielić pamięć dla nowego bufora, skopiować do niego nazwę systemowego katalogu konfiguracyjnego, a następnie połączyć ten bufor z dostarczoną nazwą pliku:

```
int open_sysconf (const char *file, int flags, int mode)
{
   const char *etc = SYSCONF_DIR; /* "/etc/" */
   char *name;

   name = alloca (strlen (etc) + strlen (file) + 1);
   strcpy (name, etc);
   strcat (name, file);
   return open (name, flags, mode);
}
```

Po powrocie z funkcji, pamięć przydzielona za pomocą funkcji `alloca()` zostaje automatycznie zwolniona, ponieważ wskaźnik stosu przesuwa się do pozycji funkcji wywołującej. Oznacza to, że nie można użyć przydzielonego obszaru pamięci po tym, gdy zakończy się funkcja używająca wywołania `alloca()`! Ponieważ nie należy wykonywać żadnego porządkowania pamięci za pomocą funkcji `free()`, ostateczny kod programu staje się trochę bardziej przejrzysty. Oto ta sama funkcja, lecz zaimplementowana przy użyciu wywołania `malloc()`:

```
int open_sysconf (const char *file, int flags, int mode)
{
   const char *etc = SYSCONF_DIR; /* "/etc/" */
   char *name;
   int fd;

   name = malloc (strlen (etc) + strlen (file) + 1);
   if (!name)
   {
      perror ("malloc");
      return -1;
   }

   strcpy (name, etc);
   strcat (name, file);
   fd = open (name, flags, mode);
   free (name);
```

```
    return fd;
}
```

Należy zauważyć, że w parametrach wywołania funkcji nie powinno używać się bezpośredniego wywołania `alloca()`. Powodem takiego zachowania jest to, że przydzielona pamięć będzie istnieć na stosie pośrodku obszaru zarezerwowanego do przechowywania parametrów funkcji. Na przykład, poniższy kod jest niepoprawny:

```
/* TAK NIE NALEŻY ROBIĆ! */
ret = foo (x, alloca (10));
```

Interfejs `alloca()` posiada ciekawą historię. W przypadku wielu systemów jego działanie było nieprawidłowe lub w pewnym sensie niezdefiniowane. W systemach posiadających nieduży stos o stałym rozmiarze, użycie funkcji `alloca()` było łatwym sposobem, aby go przepełnić i w rezultacie załamać wykonanie programu. W niektórych systemach funkcja `alloca()` nie jest do tej pory zaimplementowana. Błędne i niespójne implementacje funkcji `alloca()` spowodowały, że cieszy się ona złą reputacją.

Jeśli program powinien być przenośny, nie należy używać w nim funkcji `alloca()`. W przypadku systemu Linux funkcja ta jest jednak bardzo użytecznym i niedocenionym narzędziem. Działa wyjątkowo dobrze — w przypadku wielu architektur realizowanie przydzielania pamięci za pomocą tej funkcji nie powoduje niczego ponad zwiększenie wskaźnika stosu, dlatego też łatwo przewyższa ona pod względem wydajności funkcję `malloc()`. W przypadku niewielkich obszarów przydzielonej pamięci i kodu, specyficznego dla Linuksa, użycie funkcji `alloca()` może spowodować bardzo dobrą poprawę wydajności.

Powielanie łańcuchów znakowych na stosie

Powszechnym przykładem użycia funkcji `alloca()` jest tymczasowe powielanie łańcucha znakowego. Na przykład:

```
/* należy powielić łańcuch 'song' */
char *dup;

dup = alloca (strlen (song) + 1);
strcpy (dup, song);

/* tutaj można już używać wskaźnika 'dup'... */

return; /* 'dup' zostaje automatycznie zwolniony */
```

Z powodu częstego użycia tego rozwiązania, a również korzyści związanych z prędkością działania, jaką oferuje funkcja `alloca()`, systemy linuksowe udostępniają wersję funkcji `strdup()`, która pozwala na powielenie danego łańcucha znakowego na stosie:

```
#define _GNU_SOURCE
#include <string.h>

char * strdupa (const char *s);
char * strndupa (const char *s, size_t n);
```

Wywołanie funkcji `strdupa()` zwraca kopię łańcucha `s`. Wywołanie funkcji `strndupa()` powiela `n` znaków łańcucha `s`. Jeśli łańcuch `s` jest dłuższy od `n`, proces powielania kończy się w pozycji `n`, a funkcja dołącza na koniec skopiowanego łańcucha znak pusty. Funkcje te oferują te same korzyści co funkcja `alloca()`. Powielony łańcuch zostaje automatycznie zwolniony, gdy wywołująca funkcja kończy swoje działanie.

POSIX nie definiuje funkcji `alloca()`, `strdupa()` i `strndupa()`, a w innych systemach operacyjnych występują one sporadycznie. Jeśli należy zapewnić przenośność programu, wówczas użycie tych funkcji jest odradzane. W Linuksie wspomniane funkcje działają jednak całkiem dobrze i mogą zapewnić znakomitą poprawę wydajności, zamieniając skomplikowane czynności, związane z przydziałem pamięci dynamicznej, na zaledwie przesunięcie wskaźnika stosu.

Tablice o zmiennej długości

Standard C99 wprowadził *tablice o zmiennej długości* (ang. *variable-length arrays*, w skrócie *VLA*), których rozmiar ustalany jest podczas działania programu, a nie w czasie jego kompilacji. Kompilator GNU dla języka C wspierał takie tablice już od jakiegoś czasu, lecz odkąd standard C99 formalnie je zdefiniował, pojawił się istotny bodziec, aby ich używać. Podczas użycia tablic o zmiennej długości unika się przydzielania pamięci dynamicznej w taki sam sposób, jak podczas stosowania funkcji `alloca()`.

Sposób użycia łańcuchów o zmiennej długości jest dokładnie taki, jak się oczekuje:

```
for (i = 0; i < n; ++i)
{
   char foo[i + 1];
   /* tu można użyć 'foo'... */
}
```

W powyższym fragmencie kodu zmienna `foo` jest łańcuchem znaków o różnej długości, równej `i + 1`. Podczas każdej iteracji w pętli zostaje dynamicznie utworzona zmienna `foo`, a następnie automatycznie zwolniona, gdy znajdzie się poza zakresem widoczności. Gdyby zamiast łańcuchów o zmiennej długości użyto funkcji `alloca()`, pamięć nie zostałaby zwolniona, dopóki funkcja nie zakończyłaby swojego działania. Użycie łańcuchów o zmiennej długości zapewnia, że pamięć zostanie zwolniona podczas każdej iteracji w pętli. Dlatego też użycie takich łańcuchów zużywa w najgorszym razie `n` bajtów pamięci, podczas gdy użycie funkcji `alloca()` wykorzystywałoby `n*(n+1)/2` bajtów.

Funkcja `open_sysconf()` może zostać obecnie ponownie napisana, wykorzystując do jej implementacji łańcuch znaków o zmiennej długości:

```
int open_sysconf (const char *file, int flags, int mode)
{
   const char *etc = SYSCONF_DIR; /* "/etc/" */
   char name[strlen (etc) + strlen (file) + 1];

   strcpy (name, etc);
   strcat (name, file);

   return open (name, flags, mode);
}
```

Podstawową różnicą między użyciem funkcji `alloca()`, a użyciem tablic o zmiennej długości jest to, iż pamięć otrzymana przy użyciu tej pierwszej metody istnieje w czasie wykonywania funkcji, natomiast pamięć uzyskana przy użyciu drugiej metody istnieje do momentu, gdy zmienna, która ją reprezentuje, znajdzie się poza zakresem widoczności. Może się to zdarzyć, zanim funkcja zakończy swoje działanie — będąc cechą pozytywną lub negatywną. W przypadku pętli `for`, która została zastosowana w powyższym przykładzie, odzyskiwanie pamięci przy każdej iteracji zmniejsza realne zużycie pamięci bez żadnych efektów ubocznych (do

wykonania programu nie była potrzebna dodatkowa pamięć). Jeśli jednakże z pewnych powodów wymagane jest, aby przydzielona pamięć była dostępna dłużej niż tylko przez pojedynczą iterację pętli, wówczas bardziej sensowne jest użycie funkcji `alloca()`.

Łączenie wywołania funkcji `alloca()` oraz użycia tablic o zmiennej długości w jednym miejscu programu może powodować zaskakujące efekty. Należy postępować rozsądnie i używać tylko jednej z tych dwóch opcji w tworzonych funkcjach.

Wybór mechanizmu przydzielania pamięci

Wiele opcji przydzielania pamięci, omówionych w tym rozdziale, może być powodem powstania pytania o to, jakie rozwiązanie jest najbardziej odpowiednie dla danej czynności. W większości sytuacji użycie funkcji `malloc()` zaspokaja wszystkie potrzeby programisty. Czasami jednak inny sposób działania pozwala na uzyskanie lepszych wyników. Tabela 8.2. przedstawia informacje pomagające wybrać mechanizm przydzielania pamięci.

Tabela 8.2. Sposoby przydzielania pamięci w Linuksie

Sposób przydzielania pamięci	Zalety	Wady
Funkcja `malloc()`	Prosta, łatwa, powszechnie używana.	Pamięć zwracana nie musi być wypełniona zerami.
Funkcja `calloc()`	Prosta metoda przydzielania pamięci dla tablic, pamięć zwracana wypełniona jest zerami.	Dziwny interfejs w przypadku, gdy pamięć musi zostać przydzielona dla innych struktur danych niż tablice.
Funkcja `realloc()`	Zmienia wielkość istniejących obszarów przydzielonej pamięci.	Użyteczna wyłącznie dla operacji zmiany wielkości istniejących obszarów przydzielonej pamięci.
Funkcje `brk()` i `sbrk()`	Pozwala na szczegółową kontrolę działania sterty.	Zbyt niskopoziomowa dla większości użytkowników.
Anonimowe odwzorowania w pamięci	Łatwe w obsłudze, współdzielone, pozwalają projektantowi na ustalanie poziomu zabezpieczeń oraz dostarczania porady; optymalne rozwiązanie dla dużych przydziałów pamięci.	Niezbyt pasujące do niewielkich przydziałów pamięci; funkcja `malloc()` w razie potrzeby automatycznie używa anonimowych odwzorowań w pamięci.
Funkcja `posix_memalign()`	Przydziela pamięć wyrównaną do dowolnej, rozsądnej wartości.	Stosunkowo nowa, dlatego też jej przenośność jest dyskusyjna; użycie ma sens dopiero wówczas, gdy wyrównanie ma duże znaczenie.
Funkcje `memalign()` i `valloc()`	Bardziej popularna w innych systemach uniksowych niż funkcja `posix_memalign()`.	Nie jest zdefiniowana przez POSIX, oferuje mniejsze możliwości kontroli wyrównania niż `posix_memalign()`.
Funkcja `alloca()`	Bardzo szybki przydział pamięci, nie ma potrzeby, aby po użyciu jawnie ją zwalniać; bardzo dobra w przypadku niewielkich przydziałów pamięci.	Brak możliwości informowania o błędach, niezbyt dobra w przypadku dużych przydziałów pamięci, błędne działanie w niektórych systemach uniksowych.
Tablice o zmiennej długości	Podobnie jak `alloca()`, lecz pamięć zostanie zwolniona, gdy tablica znajdzie się poza zasięgiem widoczności, a nie podczas powrotu z funkcji.	Metoda użyteczna jedynie dla tablic; w niektórych sytuacjach może być preferowany sposób zwalniania pamięci, charakterystyczny dla funkcji `alloca()`; metoda mniej popularna w innych systemach uniksowych niż użycie funkcji `alloca()`.

Wreszcie, nie należy zapominać o alternatywie dla wszystkich powyższych opcji, czyli o automatycznym i statycznym przydziale pamięci. Przydzielanie obszarów dla zmiennych automatycznych na stosie lub dla zmiennych globalnych na stercie jest często łatwiejsze i nie wymaga obsługi wskaźników oraz troski o prawidłowe zwolnienie pamięci.

Operacje na pamięci

Język C dostarcza zbioru funkcji pozwalających bezpośrednio operować na obszarach pamięci. Funkcje te działają w wielu przypadkach w sposób podobny do interfejsów służących do obsługi łańcuchów znakowych, takich jak `strcmp()` i `strcpy()`, lecz używana jest w nich wartość rozmiaru bufora dostarczonego przez użytkownika, zamiast zakładania, że łańcuchy są zakończone znakiem zerowym. Należy zauważyć, że żadna z tych funkcji nie może zwrócić błędu. Zabezpieczenie przed powstaniem błędu jest zadaniem dla programisty — jeśli do funkcji przekazany zostanie wskaźnik do niepoprawnego obszaru pamięci, rezultatem jej wykonania nie będzie nic innego, jak tylko błąd segmentacji!

Ustawianie wartości bajtów

Wśród zbioru funkcji modyfikujących zawartość pamięci, najczęściej używana jest prosta funkcja `memset()`:

```
#include <string.h>

void * memset (void *s, int c, size_t n);
```

Wywołanie funkcji `memset()` ustawia `n` bajtów na wartość `c`, poczynając od adresu przekazanego w parametrze `s`, a następnie zwraca wskaźnik do zmienionego obszaru `s`. Funkcji używa się często, aby wypełnić dany obszar pamięci zerami:

```
/* wypełnij zerami obszar [s,s+256) */
memset (s, '\O', 256);
```

Funkcja `bzero()` jest starszym i niezalecanym interfejsem, wprowadzonym w systemie BSD w celu wykonania tej samej czynności. W nowym kodzie powinna być używana funkcja `memset()`, lecz Linux udostępnia `bzero()` w celu zapewnienia przenośności oraz wstecznej kompatybilności z innymi systemami:

```
#include <strings.h>

void bzero (void *s, size_t n);
```

Poniższe wywołanie jest identyczne z poprzednim użyciem funkcji `memset()`:

```
bzero(s, 256);
```

Należy zwrócić uwagę na to, że funkcja `bzero()`, podobnie jak inne interfejsy, których nazwy zaczynają się od litery `b`, wymaga dołączenia pliku nagłówkowego `<strings.h>`, a nie `<string.h>`.

Porównywanie bajtów

Podobnie jak ma to miejsce w przypadku użycia funkcji `strcmp()`, funkcja `memcmp()` porównuje dwa obszary pamięci, aby sprawdzić, czy są one identyczne:

Nie należy używać funkcji `memset()`, jeśli można użyć funkcji `calloc()`! Należy unikać przydzielania pamięci za pomocą funkcji `malloc()`, a następnie bezpośredniego wypełniania jej zerami przy użyciu funkcji `memset()`. Mimo że uzyska się takie same wyniki, dużo lepsze będzie użycie pojedynczego wywołania funkcji `calloc()`, która zwraca pamięć wypełnioną zerami. Nie tylko zaoszczędzi się na jednym wywołaniu funkcji, ale dodatkowo wywołanie `calloc()` będzie mogło otrzymać od jądra odpowiednio przygotowany obszar pamięci. W tym przypadku następuje uniknięcie ręcznego wypełniania bajtów zerami i poprawa wydajności.

```
#include <string.h>

int memcmp (const void *s1, const void *s2, size_t n);
```

Wywołanie tej funkcji powoduje porównanie pierwszych `n` bajtów dla obszarów pamięci `s1` i `s2` oraz zwraca `0`, jeśli bloki pamięci są sobie równe, wartość mniejszą od zera, jeśli `s1` jest mniejszy od `s2` oraz wartość większą od zera, jeśli `s1` jest większy od `s2`.

System BSD ponownie udostępnia niezalecany już interfejs, który realizuje w dużym stopniu to samo zadanie:

```
#include <strings.h>

int bcmp (const void *s1, const void *s2, size_t n);
```

Wywołanie funkcji `bmcp()` powoduje porównanie pierwszych `n` bajtów dla obszarów pamięci `s1` i `s2`, zwracając `0`, jeśli bloki są sobie równe lub wartość niezerową, jeśli są różne.

Z powodu istnienia wypełnienia struktur (opisanego wcześniej w podrozdziale Inne zagadnienia związane z wyrównaniem), porównywanie ich przy użyciu funkcji `memcmp()` lub `bcmp()` jest niepewne. W obszarze wypełnienia może istnieć niezainicjalizowany fragment nieużytecznych danych powodujący powstanie różnic podczas porównywania dwóch egzemplarzy danej struktury, które poza tym są sobie równe. Zgodnie z tym, poniższy kod nie jest bezpieczny:

```
/* czy dwa egzemplarze struktury dinghy są sobie równe? (BŁĘDNY KOD) */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
   return memcmp (a, b, sizeof (struct dinghy));
}
```

Zamiast stosować powyższe, błędne rozwiązanie, programiści, którzy muszą porównywać ze sobą struktury, powinni czynić to dla każdego elementu struktury osobno. Ten sposób pozwala na uzyskanie pewnej optymalizacji, lecz wymaga większego wysiłku niż niepewne użycie prostej funkcji `memcmp()`. Oto poprawny kod:

```
/* czy dwa egzemplarze struktury dinghy są sobie równe? */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
   int ret;

   if (a->nr_oars < b->nr_oars)
      return -1;
   if (a->nr_oars > b->nr_oars)
      return 1;

   ret = strcmp (a->boat_name, b->boat_name);
   if (ret)
      return ret;

   /* i tak dalej, dla każdego pola struktury... */
}
```

Przenoszenie bajtów

Funkcja `memmove()` kopiuje pierwszych `n` bajtów z obszaru pamięci `src` do `dst`, a następnie zwraca wskaźnik do `dst`:

```
#include <string.h>

void * memmove (void *dst, const void *src, size_t n);
```

System BSD ponownie udostępnia niezalecany już interfejs, który wykonuje tę samą czynność:

```
#include <strings.h>

void bcopy (const void *src, void *dst, size_t n);
```

Należy zwrócić uwagę na to, że mimo iż obie funkcje używają takich samych parametrów, kolejność dwóch pierwszych jest zmieniona w `bcopy()`.

Obie funkcje `bcopy()` oraz `memmove()` mogą bezpiecznie obsługiwać nakładające się obszary pamięci (na przykład, gdy część obszaru `dst` znajduje się wewnątrz `src`). Dzięki temu bajty w pamięci mogą przykładowo zostać przesunięte w stronę wyższych lub niższych adresów wewnątrz danego regionu. Ponieważ taka sytuacja jest rzadkością, a programista wiedziałby, jeśliby miała ona miejsce, dlatego też standard języka C definiuje wariant funkcji `memmove()`, który nie wspiera nakładających się rejonów pamięci. Ta wersja może działać potencjalnie szybciej:

```
#include <string.h>

void * memcpy (void *dst, const void *src, size_t n);
```

Powyższa funkcja działa identycznie jak `memmove()`, za wyjątkiem tego, że obszary `dst` i `src` nie mogą posiadać wspólnej części. Jeśli tak jest, rezultat wykonania funkcji jest niezdefiniowany.

Inną funkcją, wykonującą bezpieczne kopiowanie pamięci, jest `memccpy()`:

```
#include <string.h>

void * memccpy (void *dst, const void *src, int c, size_t n);
```

Funkcja `memccpy()` działa tak samo jak `memcpy()`, za wyjątkiem tego, że zatrzymuje proces kopiowania, jeśli wśród pierwszych `n` bajtów obszaru `src` zostanie odnaleziony bajt o wartości `c`. Funkcja zwraca wskaźnik do następnego bajta, występującego po `c` w obszarze `dst` lub `NULL`, jeśli `c` nie odnaleziono.

Ostatecznie funkcja `mempcpy()` pozwala poruszać się po pamięci:

```
#define _GNU_SOURCE
#include <string.h>

void * mempcpy (void *dst, const void *src, size_t n);
```

Funkcja `mempcpy()` działa tak samo jak `memcpy()`, za wyjątkiem tego, że zwraca wskaźnik do miejsca znajdującego się w pamięci za ostatnim skopiowanym bajtem. Jest to przydatne, gdy zbiór danych należy skopiować do następujących po sobie obszarów pamięci — nie stanowi to jednak zbyt dużego usprawnienia, ponieważ wartość zwracana jest zaledwie równa `dst + n`. Funkcja ta jest specyficzna dla GNU.

Wyszukiwanie bajtów

Funkcje `memchr()` oraz `memrchr()` wyszukują dany bajt w bloku pamięci:

```
#include <string.h>

void * memchr (const void *s, int c, size_t n);
```

Funkcja `memchr()` przeszukuje obszar pamięci o wielkości `n` bajtów, wskazywany przez parametr `s`, aby odnaleźć w nim znak `c`, który jest interpretowany jako typ `unsigned char`. Funkcja zwraca wskaźnik do miejsca w pamięci, w którym znajduje się bajt pasujący do parametru `c`. Jeśli wartość `c` nie zostanie odnaleziona, funkcja zwróci `NULL`.

Funkcja `memrchr()` działa tak samo jak funkcja `memchr()`, za wyjątkiem tego, że przeszukuje obszar pamięci o wielkości `n` bajtów, wskazywany przez parametr `s`, rozpoczynając od jego końca zamiast od początku:

```
#define _GNU_SOURCE
#include <string.h>

void * memrchr (const void *s, int c, size_t n);
```

W przeciwieństwie do `memchr()`, funkcja `memrchr()` jest rozszerzeniem GNU i nie należy do standardu języka C.

Aby przeprowadzać bardziej skomplikowane operacje wyszukiwania, można użyć funkcji o dziwnej nazwie `memmem()`, przeszukującej blok pamięci w celu odnalezienia dowolnego łańcucha bajtów:

```
#define _GNU_SOURCE
#include <string.h>

void * memmem (const void *haystack, size_t haystacklen, const void *needle,
  size_t needlelen);
```

Funkcja `memmem()` zwraca wskaźnik do pierwszego miejsca wystąpienia łańcucha bajtów `needle` o długości `needlelen`, wyrażonej w bajtach. Przeszukiwany obszar pamięci wskazywany jest przez parametr `haystack` i posiada długość `haystacklen` bajtów. Jeśli funkcja nie odnajdzie łańcucha `needle` w `haystack`, zwraca `NULL`. Jest również rozszerzeniem GNU.

Manipulowanie bajtami

Biblioteka języka C dla Linuksa dostarcza interfejsu, który pozwala na wykonywanie trywialnej operacji kodowania bajtów:

```
#define _GNU_SOURCE
#include <string.h>

void * memfrob (void *s, size_t n);
```

Wywołanie funkcji `memfrob()` koduje pierwszych `n` bajtów z obszaru pamięci wskazywanego przez `s`. Polega to na przeprowadzeniu dla każdego bajta operacji binarnej różnicy symetrycznej (XOR) z liczbą `42`. Funkcja zwraca wskaźnik do zmodyfikowanego obszaru `s`.

Aby przywrócić pierwotną zawartość zmodyfikowanego obszaru pamięci, należy dla niego ponownie wywołać funkcję `memfrob()`. Dlatego też wykonanie poniższego fragmentu kodu nie powoduje żadnych zmian w obszarze `secret`:

```
memfrob (memfrob (secret, len), len);
```

Funkcja ta nie jest jednak żadną prawdziwą (ani nawet okrojoną) namiastką operacji szyfrowania; ograniczona jest jedynie do wykonania trywialnego zaciemnienia bajtów. Jest specyficzna dla GNU.

Blokowanie pamięci

W Linuksie zaimplementowano operację *stronicowania na żądanie*, która polega na tym, że strony pobierane są z dysku w razie potrzeby, natomiast zapisywane na dysku, gdy nie są już używane. Dzięki temu nie istnieje bezpośrednie powiązanie wirtualnych przestrzeni adresowych dla procesów w systemie z całkowitą ilością pamięci fizycznej, gdyż istnienie obszaru wymiany na dysku dostarcza wrażenia posiadania prawie nieskończonej ilości tejże pamięci.

Wymiana stron wykonywana jest w sposób przezroczysty, a aplikacje w zasadzie nie muszą „interesować się" (ani nawet znać) sposobem działania stronicowania, przeprowadzanym przez jądro Linuksa. Istnieją jednak dwie sytuacje, podczas których aplikacje mogą wpływać na sposób działania stronicowania systemowego:

Determinizm
: Aplikacje, posiadające ograniczenia czasowe, wymagają deterministycznego zachowania. Jeśli pewne operacje dostępu do pamięci kończą się błędami stron (co wywołuje powstawanie kosztownych operacji wejścia i wyjścia), wówczas aplikacje te mogą przekraczać swoje parametry ograniczeń czasowych. Aby zapewnić, że wymagane strony będą zawsze znajdować się w pamięci fizycznej i nigdy nie zostaną wyrzucone na dysk, można dla danej aplikacji zagwarantować, że dostęp do pamięci nie zakończy się błędem, co pozwoli na spełnienie warunków spójności i determinizmu, a również na poprawę jej wydajności.

Bezpieczeństwo
: Jeśli w pamięci przechowywane są tajne dane prywatne, wówczas poziom bezpieczeństwa może zostać naruszony po wykonaniu operacji stronicowania i zapisaniu tych danych w postaci niezaszyfrowanej na dysku. Na przykład, jeśli prywatny klucz użytkownika jest zwykle przechowywany na dysku w postaci zaszyfrowanej, wówczas jego odszyfrowana kopia, znajdująca się w pamięci, może zostać wyrzucona do pliku wymiany. W przypadku środowiska o wysokim poziomie bezpieczeństwa, zachowanie to może być niedopuszczalne. Dla aplikacji wymagających zapewnienia dużego poziomu bezpieczeństwa, można zdefiniować, że obszar, w którym znajduje się odszyfrowany klucz, będzie istniał wyłącznie w pamięci fizycznej.

Oczywiście zmiana zachowania jądra może spowodować pogorszenie ogólnej sprawności systemu. Dla danej aplikacji nastąpi poprawa determinizmu oraz bezpieczeństwa, natomiast gdy jej strony będą zablokowane w pamięci, strony innej aplikacji będą wyrzucane na dysk. Jądro (jeśli można ufać metodzie jego zaprojektowania) zawsze optymalnie wybiera taką stronę, która powinna zostać wyrzucona na dysk (to znaczy stronę, która najprawdopodobniej nie będzie używana w przyszłości), dlatego też po zmianie jego zachowania wybór ten nie będzie już optymalny.

Blokowanie fragmentu przestrzeni adresowej

POSIX 1003.1b-1993 definiuje dwa interfejsy pozwalające na „zamknięcie" jednej lub więcej stron w pamięci fizycznej, dzięki czemu można zapewnić, że nie zostaną one nigdy wyrzucone na dysk. Pierwsza funkcja blokuje pamięć dla danego przedziału adresów:

```
#include <sys/mman.h>

int mlock (const void *addr, size_t len);
```

Wywołanie funkcji `mlock()` blokuje w pamięci fizycznej obszar pamięci wirtualnej, rozpoczynający się od adresu `addr` i posiadający wielkość `len` bajtów. W przypadku sukcesu, funkcja zwraca wartość `0`. W przypadku błędu zwraca `-1` oraz odpowiednio ustawia zmienną `errno`.

Poprawne wywołanie funkcji blokuje w pamięci wszystkie strony fizyczne, których adresy zawierają się w zakresie `[addr, addr + len)`. Na przykład, jeśli funkcja chce zablokować tylko jeden bajt, wówczas w pamięci zostanie zablokowana cała strona, w której on się znajduje. Standard POSIX definiuje, że adres `addr` powinien być wyrównany do wielkości strony. Linux nie wymusza tego zachowania i w razie potrzeby niejawnie zaokrągla adres `addr` w dół do najbliższej strony. W przypadku programów, dla których wymagane jest zachowanie warunku przenośności do innych systemów, należy jednak upewnić się, że `addr` jest wyrównany do granicy strony.

Poprawne wartości zmiennej `errno` obejmują poniższe kody błędów:

`EINVAL`
: Parametr `len` ma wartość ujemną.

`ENOMEM`
: Proces wywołujący zamierzał zablokować więcej stron, niż wynosi ograniczenie zasobów `RLIMIT_MEMLOCK` (szczegóły w podrozdziale Ograniczenia blokowania).

`EPERM`
: Wartość ograniczenia zasobów `RLIMIT_MEMLOCK` była równa zeru, lecz proces nie posiadał uprawnienia `CAP_IPC_LOCK` (podobnie, szczegóły w podrozdziale Ograniczenia blokowania).

Podczas wykonywania funkcji `fork()`, proces potomny nie dziedziczy pamięci zablokowanej. Dzięki istnieniu mechanizmu kopiowania podczas zapisu, używanego dla przestrzeni adresowych w Linuksie, strony procesu potomnego są skutecznie zablokowane w pamięci, dopóki potomek nie wykona dla nich operacji zapisu.

Załóżmy przykładowo, że pewien program przechowuje w pamięci odszyfrowany łańcuch znaków. Proces może za pomocą kodu, podobnego do poniżej przedstawionego, zablokować stronę zawierającą dany łańcuch:

```
int ret;

/* zablokuj łańcuch znaków 'secret' w pamięci */
ret = mlock (secret, strlen (secret));
if (ret)
   perror ("mlock");
```

Blokowanie całej przestrzeni adresowej

Jeśli proces wymaga zablokowania całej przestrzeni adresowej w pamięci fizycznej, wówczas użycie funkcji `mlock()` staje się niewygodne. Aby zrealizować to zadanie — powszechnie wykonywane w przypadku aplikacji czasu rzeczywistego — standard POSIX definiuje funkcję systemową, która blokuje całą przestrzeń adresową:

```
#include <sys/mman.h>

int mlockall (int flags);
```

Wywołanie funkcji mlockall() blokuje w pamięci fizycznej wszystkie strony przestrzeni adresowej dla aktualnego procesu. Parametr flags steruje zachowaniem funkcji i jest równy sumie bitowej poniższych znaczników:

MCL_CURRENT
: Jeśli znacznik jest ustawiony, powoduje to, że funkcja mlockall() blokuje wszystkie aktualnie odwzorowane strony w przestrzeni adresowej procesu. Stronami takimi może być stos, segment danych, pliki odwzorowane itd.

MCL_FUTURE
: Jeśli znacznik jest ustawiony, wówczas wykonanie funkcji mlockall() zapewnia, iż wszystkie strony, które w przyszłości zostaną odwzorowane w przestrzeni adresowej, będą również zablokowane w pamięci.

Większość aplikacji używa obu tych znaczników jednocześnie.

W przypadku sukcesu funkcja zwraca wartość 0. W przypadku błędu zwraca −1 oraz odpowiednio ustawia zmienną errno na jedną z poniższych wartości:

EINVAL
: Parametr flags ma wartość ujemną.

ENOMEM
: Proces wywołujący zamierzał zablokować więcej stron, niż wynosi ograniczenie zasobów RLIMIT_MEMLOCK (szczegóły w podrozdziale Ograniczenia blokowania).

EPERM
: Wartość ograniczenia zasobów RLIMIT_MEMLOCK była równa zeru, lecz proces nie posiadał uprawnienia CAP_IPC_LOCK (podobnie, szczegóły w podrozdziale Ograniczenia blokowania).

Odblokowywanie pamięci

Aby umożliwić odblokowanie stron z pamięci fizycznej, pozwalając jądru w razie potrzeby ponownie wyrzucać je na dysk, POSIX definiuje dwa dodatkowe interfejsy:

```
#include <sys/mman.h>

int munlock (const void *addr, size_t len);
int munlockall (void);
```

Funkcja systemowa munlock() odblokowuje strony, które rozpoczynają się od adresu addr i zajmują obszar len bajtów. Jest ona przeciwieństwem funkcji mlock(). Funkcja systemowa munlockall() jest przeciwieństwem mlockall(). Obie funkcje zwracają zero w przypadku sukcesu, natomiast w przypadku niepowodzenia zwracają −1 oraz ustawiają zmienną errno na jedną z poniższych wartości:

EINVAL
: Parametr len jest nieprawidłowy (tylko dla munlock()).

ENOMEM
: Niektóre z podanych stron są nieprawidłowe.

EPERM
: Wartość ograniczenia zasobów RLIMIT_MEMLOCK była równa zeru, lecz proces nie posiadał uprawnienia CAP_IPC_LOCK (szczegóły w następnym podrozdziale Ograniczenia blokowania).

Blokady pamięci nie zagnieżdżają się. Dlatego też, bez względu na to, ile razy dana strona została zablokowana za pomocą funkcji `mlock()` lub `mlockall()`, pojedyncze wywołanie funkcji `munlock()` lub `munlockall()` spowoduje jej odblokowanie.

Ograniczenia blokowania

Ponieważ blokowanie pamięci może spowodować spadek wydajności systemu (faktycznie, jeśli zbyt wiele stron zostanie zablokowanych, operacje przydziału pamięci mogą się nie powieść), dlatego też w systemie Linux zdefiniowano ograniczenia, które określają, ile stron może zostać zablokowanych przez jeden proces.

Proces, który posiada uprawnienie `CAP_IPC_LOCK`, może zablokować dowolną liczbę stron w pamięci. Procesy nieposiadające takiego uprawnienia, mogą zablokować wyłącznie tyle bajtów pamięci, ile wynosi ograniczenie `RLIMIT_MEMLOCK`. Domyślnie, ograniczenie to wynosi 32 kB — jest ono wystarczające, aby zablokować jeden lub dwa tajne klucze w pamięci, lecz nie tak duże, aby skutecznie wpłynąć na wydajność systemu (w rozdziale 6. omówiono ograniczenia zasobów oraz metody pozwalające na pobieranie i ustawianie tych parametrów).

Czy strona znajduje się w pamięci fizycznej?

Aby ułatwić uruchamianie programów oraz usprawnić diagnostykę, Linux udostępnia funkcję `mincore()`, która może zostać użyta, by ustalić, czy obszar danych znajduje się w pamięci fizycznej lub w pliku wymiany na dysku:

```
#include <unistd.h>
#include <sys/mman.h>

int mincore (void *start, size_t length, unsigned char *vec);
```

Wywołanie funkcji `mincore()` zwraca wektor bajtów, który opisuje, jakie strony odwzorowania znajdują się w pamięci fizycznej w czasie jej użycia. Funkcja zwraca wektor poprzez parametr `vec` oraz opisuje strony rozpoczynające się od adresu `start` (który musi być wyrównany do granicy strony) i obejmujące obszar o wielkości `length` bajtów (który nie musi być wyrównany do granicy strony). Każdy element w wektorze `vec` odpowiada jednej stronie z dostarczonego zakresu adresów, poczynając od pierwszego bajta opisującego pierwszą stronę i następnie przechodząc w sposób liniowy do kolejnych stron. Zgodnie z tym, wektor `vec` musi być na tyle duży, aby przechować odpowiednią liczbę bajtów, równą wyrażeniu `(length - 1 + rozmiar strony)/rozmiar strony`. Najmniej znaczący bit w każdym bajcie wektora równy jest 1, gdy strona znajduje się w pamięci fizycznej lub 0, gdy jej tam nie ma. Inne bity są obecnie niezdefiniowane i zarezerwowane do przyszłego wykorzystania.

W przypadku sukcesu funkcja zwraca `0`. W przypadku błędu zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EAGAIN`
: Brak wystarczających zasobów jądra, aby zakończyć tę operację.

`EFAULT`
: Parametr `vec` wskazuje na błędny adres.

`EINVAL`
: Parametr `start` nie jest wyrównany do granicy strony.

ENOMEM
: Obszar `[start, start + length)` zawiera pamięć, która nie jest częścią odwzorowania opartego na pliku.

Ta funkcja systemowa działa obecnie poprawnie jedynie dla odwzorowań opartych na plikach i utworzonych za pomocą opcji `MAN_SHARED`. Bardzo ogranicza to jej zakres użycia.

Przydział oportunistyczny

W systemie Linux używana jest strategia *przydziału oportunistycznego*. Gdy proces żąda przydzielenia mu dodatkowej pamięci z jądra — na przykład, poprzez zwiększenie segmentu danych lub stworzenie nowego odwzorowania w pamięci — wówczas jądro *zatwierdza* przyjęcie zlecenia na przydział pamięci, ale bez rzeczywistego dostarczenia dodatkowego fizycznego miejsca. Dopiero wówczas, gdy proces wykonuje operację zapisu dla nowo przydzielonej pamięci, jądro *realizuje* przydział poprzez zamianę zlecenia na fizyczne udostępnienie pamięci. Strategia ta jest zaimplementowana dla każdej strony z osobna, a jądro wykonuje wymagane operacje stronicowania oraz kopiowania podczas zapisu tylko w razie potrzeby.

Zachowanie to ma dużo zalet. Po pierwsze, strategia leniwego przydziału pamięci pozwala jądru przesuwać wykonywanie czynności na ostatni dopuszczalny moment, jeśli w ogóle zaistnieje potrzeba realizacji operacji przydziału. Po drugie, ponieważ żądania realizowane są dla każdej strony z osobna i wyłącznie w razie potrzeby, dlatego też tylko ta pamięć, która jest rzeczywiście używana, wykorzystuje zasoby fizyczne. Wreszcie, ilość pamięci zatwierdzonej może być dużo większa od ilości pamięci fizycznej, a nawet od dostępnego obszaru wymiany. Ta ostatnia cecha zwana jest *przekroczeniem zakresu zatwierdzenia* (ang. *overcommitment*).

Przekroczenie zakresu zatwierdzenia oraz stan braku pamięci (OOM)

Przekroczenie zakresu zatwierdzenia pozwala systemom na uruchamianie dużo większej liczby obszerniejszych aplikacji, niż byłoby to możliwe, gdyby każda żądana strona pamięci otrzymywała odwzorowanie w zasobie fizycznym w momencie jej przydziału zamiast w momencie użycia. Bez mechanizmu przekraczania zakresu zatwierdzenia, wykonanie odwzorowania pliku o wielkości 2 GB przy użyciu kopiowania podczas zapisu, wymagałoby od jądra przydzielenia 2 GB pamięci fizycznej. Dzięki mechanizmowi przekraczania zakresu zatwierdzenia, odwzorowanie pliku 2 GB wymaga przydzielenia pamięci fizycznej jedynie dla poszczególnych stron z danymi, które są faktycznie zapisywane przez proces. Ponadto, bez użycia mechanizmu przekraczania zakresu zatwierdzenia, każde wywołanie funkcji `fork()` wymagałoby dostarczenia odpowiednio dużego obszaru wolnej pamięci, aby móc skopiować przestrzeń adresową, nawet gdyby większość stron nie zostało poddanych operacji kopiowania podczas zapisu.

Co stanie się jednak, gdy procesy przystąpią do realizacji zaległych przydziałów, których sumaryczna wielkość przekroczy rozmiar pamięci fizycznej i obszaru wymiany? W tym przypadku jedna lub więcej realizacji przydziału zakończy się niepowodzeniem. Ponieważ jądro zrealizowało już przydział pamięci — wykonanie funkcji systemowej, żądającej przeprowadzenia tej operacji, zakończyło się sukcesem — a proces właśnie przystępuje do użycia udostępnionej pamięci, dlatego też jedyną dostępną opcją jądra jest przerwanie działania tego procesu i zwolnienie zajętej przez niego pamięci.

Gdy przekroczenie zakresu zatwierdzenia powoduje pojawienie się niewystarczającej ilości pamięci, aby zatwierdzić zrealizowane żądanie, wówczas sytuację taką nazywa się *stanem braku pamięci* (ang. *out of memory*, w skrócie *OOM*). W odpowiedzi na taką sytuację jądro uruchamia *zabójcę stanu braku pamięci* (ang. *OOM killer*), aby wybrał proces, który „nadaje się" do usunięcia. W tym celu jądro próbuje odnaleźć najmniej ważny proces, zużywający największą ilość pamięci.

Stany braku pamięci występują rzadko, dlatego też odnosi się duże korzyści z zezwolenia na przekraczanie zakresu zatwierdzenia. Na pewno jednak pojawienie się takiego stanu nie jest mile widziane, a niedeterministyczne przerwanie działania procesu przez zabójcę OOM jest często nie do zaakceptowania.

W systemach, których to dotyczy, jądro pozwala na zablokowanie przekraczania zakresu zatwierdzenia przy użyciu pliku */proc/sys/vm/overcommit_memory* oraz analogicznego parametru *sysctl* o nazwie `vm.overcommit_memory`.

Domyślna wartość tego parametru równa jest zeru i nakazuje ona jądru, aby realizował heurystyczną strategię przekraczania zakresu zatwierdzenia, która pozwala na zatwierdzanie przydziałów pamięci w granicach rozsądku, nie dopuszczając jednak do realizacji wyjątkowo złych żądań. Wartość równa `1` pozwala na wykonanie wszystkich zatwierdzeń, podejmując ryzyko powstania stanu braku pamięci. Pewne aplikacje, wykorzystujące zasoby pamięci w intensywny sposób, takie jak programy naukowe, próbują wysyłać tak dużo żądań przydziału pamięci, które i tak nigdy nie będą musiały zostać zrealizowane, że użycie tej opcji ma w tym przypadku sens.

Wartość równa `2` całkowicie uniemożliwia przekraczanie zakresu zatwierdzenia i aktywuje *rozliczanie ścisłe* (ang. *strict accounting*). W tym trybie zatwierdzenia przyjęcia zleceń przydziału pamięci ograniczone są do wielkości obszaru wymiany oraz pewnego fragmentu pamięci fizycznej, którego względny rozmiar jest konfigurowalny. Rozmiar ten można ustawić za pomocą pliku */proc/sys/vm/overcommit_ratio* lub analogicznego parametru *sysctl*, zwanego `vm.overcommit_ratio`. Domyślną wartością jest liczba `50`, ograniczająca zatwierdzenia przyjęcia zleceń przydziału pamięci do wielkości obszaru wymiany i połowy pamięci fizycznej. Ponieważ pamięć fizyczna zawiera jądro, tablice stron, strony zarezerwowane przez system, strony zablokowane itd., dlatego też tylko jej fragment może być w rzeczywistości wyrzucany na dysk i realizować przydziały pamięci.

Rozliczenia ścisłego należy używać z rozwagą! Wielu projektantów systemowych, którzy są zniechęceni opiniami o zabójcy OOM, uważa, że użycie rozliczenia ścisłego spowoduje rozwiązanie ich problemów. Aplikacje często jednak wykonują mnóstwo niepotrzebnych przydziałów pamięci, które sięgają daleko poza obszar przekraczania zakresu zatwierdzenia. Akceptacja takiego zachowania była jednym z argumentów przemawiających za implementacją pamięci wirtualnej.

ROZDZIAŁ 9.

Sygnały

Sygnały są przerwaniami programowymi, które dostarczają mechanizmu pozwalającego na obsługę zdarzeń asynchronicznych. Zdarzenia te mogą powstawać poza systemem (na przykład, gdy użytkownik wysyła znak przerwania — uzyskiwany zwykle poprzez naciśnięcie sekwencji klawiszy *Ctrl+C*) lub być generowane przez program lub jądro — na przykład, gdy proces wykonuje kod przeprowadzający operację dzielenia przez zero. Proces może również wysyłać sygnały do innego procesu i przez to realizować prymitywną odmianę komunikacji międzyprocesorowej (IPC).

Punktem kluczowym całego zagadnienia nie jest jedynie asynchroniczne występowanie zdarzeń (użytkownik może na przykład wysłać sekwencję klawiszy *Ctrl+C* w każdym momencie działania programu), lecz również zdolność programu do asynchronicznej obsługi sygnałów. Funkcje obsługujące sygnały są zarejestrowane w jądrze. Z chwilą nadejścia sygnałów jądro wywołuje te funkcje w sposób asynchroniczny do pozostałej części programu.

Sygnały od dawna były częścią systemu Unix. Z biegiem czasu ewoluowały w kategoriach niezawodności (dawniej „znikały" w systemie) oraz funkcjonalności (obecnie mogą zawierać pola zdefiniowane przez użytkownika). Na początku w różnych systemach uniksowych przeprowadzono niekompatybilne zmiany w opisach sygnałów. Na szczęście POSIX uratował sytuację i zdefiniował standard ich obsługi, który jest wspierany w systemie Linux i zostanie tutaj omówiony.

Na początku tego rozdziału przedstawiony zostanie przegląd sygnałów oraz analiza poświęcona właściwym i niewłaściwym sposobom ich użycia. Następnie zaprezentowane zostaną różne interfejsy Linuksa, dzięki którym można obsługiwać sygnały i zarządzać nimi.

Sygnałów używa się w większości złożonych aplikacji. Nawet jeśli dana aplikacja jest celowo zaprojektowana tak, aby w swojej warstwie komunikacyjnej nie używała sygnałów (jest to często dobrym pomysłem!), mimo to w niektórych przypadkach wciąż istnieje konieczność współpracy z sygnałami, na przykład podczas obsługi procesu kończenia działania programu.

Koncepcja sygnałów

Sygnały posiadają określony czas istnienia. Sygnał zostaje *zgłoszony* (*wysłany* lub *wygenerowany*), a następnie *przechowywany* przez jądro, dopóki nie będzie mogło go dostarczyć. Gdy pojawia się taka możliwość, jądro w odpowiedni sposób *obsługuje* sygnał. W zależności od tego, jakie wymagania posiada proces, jądro może wykonać jedną z poniżej przedstawionych trzech czynności:

Zignorować *sygnał*
: Nie jest podejmowana żadna akcja. Istnieją dwa sygnały, które nie mogą zostać zignorowane: `SIGKILL` oraz `SIGSTOP`. Administrator systemu musi posiadać możliwość usuwania lub zatrzymywania procesów, bo gdyby proces mógł zignorować sygnał `SIGKILL` (przez co nie można byłoby go usunąć) lub sygnał `SIGSTOP` (dzięki czemu proces stałby się niemożliwy do zatrzymania), wówczas byłoby to naruszeniem powyższej zasady.

Przechwycić *i* obsłużyć *sygnał*
: Jądro wstrzyma wykonywanie aktualnej ścieżki kodu dla danego procesu i wywoła wcześniej zarejestrowaną funkcję. Następnie proces wykona tę funkcję. Gdy tylko zakończy się jej działanie, proces ponownie wróci do poprzedniego miejsca w kodzie, w którym znajdował się w momencie przechwycenia sygnału.

Sygnały `SIGINT` oraz `SIGTERM` są dwoma powszechnie przechwytywanymi sygnałami. Sygnał `SIGINT` jest przechwytywany przez procesy, aby obsłużyć wygenerowanie znaku przerwania przez użytkownika — na przykład, możliwe byłoby przechwycenie tego sygnału w terminalu i powrót do standardowego wyświetlania znaku zachęty w linii poleceń. Sygnał `SIGTERM` jest przejmowany przez procesy, aby przed zakończeniem programu przeprowadzić niezbędne operacje porządkujące, takie jak odłączenie od sieci lub usunięcie plików tymczasowych. Sygnały `SIGKILL` i `SIGSTOP` nie mogą zostać przechwycone.

Wykonać akcję domyślną
: Akcja ta jest zależna od wysyłanego sygnału. Domyślną akcją jest często przerwanie działania procesu. Dotyczy to na przykład sygnału `SIGKILL`. Wiele sygnałów związanych jest jednak ze specyficznymi zastosowaniami, które interesują programistów tylko w pewnych sytuacjach, dlatego też takie sygnały są domyślnie ignorowane, ponieważ w wielu programach nie istnieje potrzeba, aby się nimi zajmować. Wkrótce zostaną omówione różne sygnały i odpowiednie dla nich akcje domyślne.

Dawniej, gdy sygnał został dostarczony, funkcja, która go obsługiwała, nie posiadała żadnej informacji na temat powodów jego wygenerowania, za wyjątkiem tego, że został zgłoszony określony rodzaj sygnału. Obecnie jądro dostarcza mnóstwo informacji kontekstowej dla programistów, którzy chcą ją wykorzystać, a sygnały mogą nawet przenosić dane zdefiniowane przez użytkownika, jak się to dzieje w przypadku mniej lub bardziej zaawansowanych mechanizmów IPC.

Identyfikatory sygnałów

Każdy sygnał posiada nazwę symboliczną, która rozpoczyna się od przedrostka `SIG`. Na przykład `SIGINT` jest sygnałem, który zostaje wysłany, gdy użytkownik naciska klawisze *Ctrl+C*, `SIGABRT` pojawia się, gdy proces wywołuje funkcję `abort()`, a `SIGKILL` zostaje wysłany, gdy proces jest zmuszony do przerwania swojego działania.

Wszystkie te sygnały są zdefiniowane w pliku nagłówkowym dołączanym do pliku `<signal.h>`. Są zwykłymi definicjami preprocesora, które reprezentują dodatnie liczby całkowite — oznacza to, że każdy sygnał związany jest również z liczbowym identyfikatorem. Odwzorowanie nazwy na liczbę całkowitą w przypadku sygnałów jest zależne od implementacji i zmienia się w zależności od rodzaju systemu uniksowego, choć początkowe sygnały są zazwyczaj odwzorowane w taki sam sposób (na przykład, `SIGKILL` to cieszący się złą sławą *sygnał o numerze 9*). Dobry programista będzie zawsze używał nazwy symbolicznej sygnału zamiast jego wartości liczbowej.

Numery sygnałów rozpoczynają się od `1` (jest nim zazwyczaj `SIGHUP`) i rosną w sposób liniowy. W sumie istnieje około 31 sygnałów, lecz w większości programów obsługuje się tylko kilka z nich. Nie istnieje sygnał o wartości `0`, lecz jest specjalna wartość nazywana *sygnałem zerowym*. Nie jest on jednak żadnym ważnym sygnałem (nie zasługuje na specjalną nazwę), choć w wyjątkowych przypadkach jest używany przez niektóre funkcje systemowe (na przykład `kill()`).

Listę sygnałów, wspieranych w danym systemie, można wygenerować przez użycie polecenia *kill -l*.

Sygnały wspierane przez system Linux

Tabela 9.1. przedstawia listę sygnałów, które są wspierane przez system Linux.

Tabela 9.1. Sygnały

Sygnał	Opis	Akcja domyślna
`SIGABRT`	Wysyłany przez funkcję `abort()`.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
`SIGALRM`	Wysyłany przez funkcję `alarm()`.	Przerwanie procesu.
`SIGBUS`	Błąd sprzętu lub wyrównania.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
`SIGCHLD`	Proces potomny został zakończony.	Ignorowanie sygnału.
`SIGCONT`	Zatrzymany proces ponownie rozpoczyna swoje działanie.	Ignorowanie sygnału.
`SIGFPE`	Wyjątek arytmetyczny.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
`SIGHUP`	Sterujący terminal procesu został zamknięty (najprawdopodobniej użytkownik wylogował się z systemu).	Przerwanie procesu.
`SIGILL`	Proces próbował wykonać niedopuszczalną instrukcję.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
`SIGINT`	Użytkownik wysłał znak przerwania (*Ctrl+C*).	Przerwanie procesu.
`SIGIO`	Zdarzenie asynchronicznej operacji wejścia i wyjścia.	Przerwanie procesu[1].
`SIGKILL`	Bezwarunkowe przerwanie działania procesu.	Przerwanie procesu.
`SIGPIPE`	Proces zapisywał do potoku, lecz nie istnieli odbiorcy wiadomości.	Przerwanie procesu.
`SIGPROF`	Upłynął czas dla licznika profilującego.	Przerwanie procesu.

[1] W przypadku innych systemów uniksowych, takich jak BSD, zachowaniem domyślnym jest ignorowanie tego sygnału.

Tabela 9.1. Sygnały — ciąg dalszy

Sygnał	Opis	Akcja domyślna
SIGPWR	Awaria zasilania.	Przerwanie procesu.
SIGQUIT	Użytkownik wysłał znak wyjścia (*Ctrl+*).	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGSEGV	Błąd dostępu do pamięci.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGSTKFLT	Błąd stosu koprocesora.	Przerwanie procesu[2].
SIGSTOP	Zatrzymanie wykonania procesu.	Zatrzymanie procesu.
SIGSYS	Proces próbował wykonać błędną funkcję systemową.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGTERM	Przerwanie wykonywania procesu, możliwe do przechwycenia.	Przerwanie procesu.
SIGTRAP	Napotkano punkt wstrzymania.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGTSTP	Użytkownik wysłał znak zatrzymania (*Ctrl+Z*).	Zatrzymanie procesu.
SIGTTIN	Proces drugorzędny czytał z terminala sterującego.	Zatrzymanie procesu.
SIGTTOU	Proces drugorzędny pisał do terminala sterującego.	Zatrzymanie procesu.
SIGURG	Pilna operacja wejścia i wyjścia oczekuje na obsługę.	Ignorowanie sygnału.
SIGUSR1	Sygnał zdefiniowany dla procesu.	Przerwanie procesu.
SIGUSR2	Sygnał zdefiniowany dla procesu.	Przerwanie procesu.
SIGVTALRM	Sygnał wygenerowany przez funkcję `setitimer()` podczas jej wywołania, z ustawionym znacznikiem `ITIMER_VIRTUAL`.	Przerwanie procesu.
SIGWINCH	Rozmiar okna dla terminala sterującego uległ zmianie.	Ignorowanie sygnału.
SIGXCPU	Przekroczenie ograniczeń dla zasobów procesora.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGXFSZ	Przekroczenie ograniczeń dla zasobów plikowych.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.

Istnieją też inne sygnały, lecz w systemie Linux są one zdefiniowane jako równoważniki istniejących wartości: `SIGINFO` jest zdefiniowany jako `SIGPWR`[3], `SIGIOT` jako `SIGABRT`, a `SIGPOLL` oraz `SIGLOST` jako `SIGIO`.

W powyższej tabeli przedstawiono skróconą definicję sygnałów, teraz dokładnie je scharakteryzujemy:

`SIGABRT`
: Sygnał ten zostaje wysłany przez funkcję `abort()` do procesu, który ją wywołuje. Proces kończy swoje działanie oraz generuje plik zrzutu systemowego. Funkcje weryfikacji warunków, takie jak `assert()`, wywołują w Linuksie `abort()`, gdy warunek nie jest spełniony.

`SIGALRM`
: Sygnał ten jest wysyłany przez funkcje `alarm()` oraz `setitimer()` (z aktywnym znacznikiem `ITIMER_REAL`) do procesu, który je wywołał, gdy upłynie termin ważności dla alarmu. Te i podobne im funkcje omówione zostaną w rozdziale 10.

[2] Jądro Linuksa nie generuje już tego sygnału; pozostał on jedynie w celu zachowania wstecznej kompatybilności.

[3] Sygnał ten jest zdefiniowany wyłącznie w architekturze Alpha. W przypadku innych architektur maszynowych `SIGPWR` nie istnieje.

SIGBUS
: Jądro zgłasza ten sygnał, gdy wykonanie procesu powoduje powstanie błędu sprzętowego, innego niż błąd ochrony pamięci, który generuje SIGSEGV. W tradycyjnych systemach uniksowych sygnał ten reprezentował różne błędy, niemożliwe do naprawienia, takie jak dostęp do pamięci niewyrównanej. Jądro Linuksa naprawia jednak większość z nich w sposób automatyczny i nie generuje SIGBUS. Jądro nie zgłasza tego sygnału, gdy proces w sposób niepoprawny próbuje uzyskać dostęp do rejonu pamięci stworzonego za pomocą funkcji mmap() (analiza odwzorowań w pamięci przeprowadzona jest w rozdziale 8.). Dopóki ten sygnał nie będzie przechwycony, jądro zakończy działanie procesu i wygeneruje plik zrzutu systemowego.

SIGCHLD
: Gdy proces kończy swoje działanie lub zatrzymuje się, jądro wysyła ten sygnał do jego procesu rodzicielskiego. Ponieważ SIGCHLD jest domyślnie ignorowany, procesy muszą go jawnie przechwycić i obsłużyć, jeśli są zainteresowane tym, co dzieje się z ich potomkami. Procedura obsługi tego sygnału wywołuje zazwyczaj funkcję wait(), omówioną w rozdziale 5., aby ustalić identyfikator procesu oraz kod powrotu dla potomka.

SIGCONT
: Sygnał ten zostaje wysłany przez jądro do procesu, który był zatrzymany, a obecnie ponownie rozpoczyna swoje wykonywanie. SIGCONT jest domyślnie ignorowany, lecz może zostać przechwycony przez procesy, jeśli wymagana jest jakaś akcja po ponownym rozpoczęciu ich wykonywania. Sygnał ten jest powszechnie używany przez terminale lub edytory, które wymagają uaktualnienia zawartości ekranu.

SIGFPE
: Wbrew swojej nazwie sygnał ten reprezentuje powstanie dowolnego wyjątku arytmetycznego, niekoniecznie związanego z operacjami zmiennoprzecinkowymi. Rodzajami wyjątków są przepełnienia, niedomiary lub dzielenie przez zero. Domyślną akcją jest przerwanie procesu i wygenerowanie pliku zrzutu systemowego, lecz procesy mogą przechwycić i obsłużyć ten sygnał, jeśli zaistnieje taka potrzeba. Należy zauważyć, że zachowanie procesu oraz wynik operacji, która spowodowała powstanie problemu, będą niezdefiniowane, jeśli proces postanowi kontynuować swoje działanie.

SIGHUP
: Sygnał ten zostaje wysłany przez jądro do lidera sesji, gdy terminal tej sesji zostaje odłączony. Jądro wysyła ten sygnał również do każdego procesu z pierwszoplanowej grupy procesów, gdy lider sesji kończy swoje działanie. Domyślną akcją jest przerwanie działania procesu i ma to sens — sygnał informuje, że użytkownik się wylogował. Procesy demonów „przesłaniają" ten sygnał mechanizmem, który umożliwia ponowne załadowanie ich plików konfiguracyjnych. Wysłanie sygnału SIGHUP na przykład do serwera Apache spowoduje, że odczytany zostanie ponownie plik *httpd.conf*. Użycie do tego celu sygnału SIGHUP jest powszechnie stosowaną konwencją, lecz nie jest wymagane. To bezpieczny sposób, ponieważ demony nie posiadają terminali sterujących i dlatego też nigdy nie powinny odebrać takiego sygnału.

SIGILL
: Sygnał ten zostaje wysłany przez jądro, gdy proces przystępuje do wykonania niepoprawnej instrukcji maszynowej. Domyślną akcją jest przerwanie działania procesu i wygenerowanie pliku zrzutu systemowego. Procesy mogą próbować przechwycić oraz obsłużyć sygnał SIGILL, lecz ich zachowanie będzie wówczas niezdefiniowane.

`SIGINT`
: Sygnał ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy użytkownik generuje znak przerwania (zwykle *Ctrl+C*). Domyślnym zachowaniem jest przerwanie działania procesu, lecz sygnał ten może zostać przez niego przechwycony i obsłużony. Zazwyczaj dzieje się tak, aby wykonać porządkowanie przed zakończeniem działania procesu.

`SIGIO`
: Sygnał ten zostaje wysłany, gdy wygenerowane jest asynchroniczne zdarzenie operacji wejścia i wyjścia w stylu BSD. Ten sposób wykonywania operacji wejścia i wyjścia jest rzadko używany w systemie Linux (w rozdziale 4. omówione zostały zaawansowane techniki wykonywania operacji wejścia i wyjścia, które są charakterystyczne dla Linuksa).

`SIGKILL`
: Sygnał ten zostaje wysłany z funkcji systemowej `kill()`; dostarcza administratorom systemu sprawdzonego sposobu pozwalającego na bezwarunkowe przerwanie działania procesu. Sygnał ten nie może zostać przechwycony ani zignorowany, a jego wynikiem jest zawsze zakończenie procesu.

`SIGPIPE`
: Jeśli proces wykonuje operację zapisu do potoku, lecz odbiorca danych zakończył swoje działanie, wówczas sygnał ten zostaje zgłoszony przez jądro. Domyślną akcją jest przerwanie działania procesu, lecz sygnał ten może zostać przez niego przechwycony i obsłużony.

`SIGPROF`
: Sygnał ten zostaje wygenerowany przez funkcję `setitimer()`, użytą ze znacznikiem `ITIMER_PROF`, gdy upłynął czas dla licznika profilującego. Domyślną akcją jest przerwanie działania procesu.

`SIGPWR`
: Ten sygnał jest zależny od systemu. W przypadku Linuksa reprezentuje on niski stan naładowania urządzenia podtrzymującego napięcie (takiego jak zasilacz awaryjny). Demon śledzący stan zasilacza awaryjnego wysyła ten sygnał do procesu *init*, który odpowiada na niego poprzez wykonanie operacji porządkowania i zamknięcia systemu operacyjnego — zakładając, że nastąpi to przed całkowitym zanikiem zasilania!

`SIGQUIT`
: Sygnał ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy użytkownik generuje znak wyjścia z terminala (zwykle *Ctrl+*). Domyślną akcją jest przerwanie działania procesu i wygenerowanie pliku zrzutu systemowego.

`SIGSEGV`
: Sygnał ten, którego nazwa oznacza *błąd segmentacji*, wysyłany jest do procesu, który pragnął uzyskać dostęp do niepoprawnego adresu pamięci. Dotyczy to również dostępu do nieodwzorowanej pamięci, czytania z pamięci, która nie posiada uprawnienia do odczytu, uruchamiania kodu w pamięci pozbawionej uprawnienia do wykonywania lub zapisywania do pamięci, która nie posiada uprawnienia do zapisu. Sygnał ten może zostać przechwycony i obsłużony przez dany proces, lecz domyślną akcją jest przerwanie jego działania i wygenerowanie pliku zrzutu systemowego.

SIGSTOP
: Sygnał ten jest wysyłany przez funkcję `kill()`. Zatrzymuje on bezwarunkowo proces i nie może zostać przechwycony ani zignorowany.

SIGSYS
: Sygnał ten zostaje wysłany przez jądro do procesu, gdy próbuje on wywołać niepoprawną funkcję systemową. Może się to zdarzyć, gdy plik binarny zostanie stworzony w nowszej wersji systemu operacyjnego (z nowszymi wersjami funkcji systemowych), lecz będzie uruchomiony w maszynie, która wyposażona jest w starszą wersję systemu. Poprawnie stworzone pliki binarne, które wywołują funkcję systemowe poprzez bibliotekę *glibc*, nie powinny nigdy otrzymać tego sygnału. Błędna funkcja systemowa powinna zwrócić `-1` oraz ustawić zmienną `errno` na `ENOSYS`.

SIGTERM
: Sygnał ten jest wysyłany wyłącznie przez funkcję `kill()`; pozwala użytkownikowi na poprawne zakończenie działania procesu (jest to akcja domyślna). Sygnał ten może zostać przechwycony przez dany proces, dzięki czemu możliwe będzie wykonanie operacji porządkowania, lecz niepoprawnym działaniem jest przechwycenie sygnału, a następnie przeprowadzenie niewłaściwego zakończenia procesu.

SIGTRAP
: Sygnał ten zostaje wysłany przez jądro do procesu, gdy napotyka on punkt wstrzymania. Zasadniczo `SIGTRAP` ostaje przechwycony przez programy wspomagające uruchamianie, a inne procesy go ignorują.

SIGTSTP
: Sygnał ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy użytkownik generuje znak zatrzymania (zwykle *Ctrl+Z*).

SIGTTIN
: Sygnał ten zostaje wysłany do procesu działającego w tle, gdy zamierza on czytać z terminala sterującego. Domyślną akcją jest zatrzymanie tego procesu.

SIGTTOU
: Sygnał ten zostaje wysłany do procesu, działającego w tle, gdy zamierza on pisać do terminala sterującego. Domyślną akcją jest zatrzymanie tego procesu.

SIGURG
: Sygnał ten zostaje wysłany do procesu, gdy w gnieździe pojawiły się dane poza kolejką. Analiza danych poza kolejką nie jest uwzględniona w tej książce.

SIGUSR1 oraz SIGUSR2
: Sygnały te są udostępnione dla zastosowań zdefiniowanych przez użytkownika: nie zostają one nigdy zgłoszone przez jądro. Procesy mogą używać sygnałów `SIGUSR1` oraz `SIGUSR2` dla dowolnych celów. Powszechnym sposobem ich użycia jest wysłanie instrukcji do demona, aby zmienić jego sposób działania. Domyślną akcją jest przerwanie działania procesu.

SIGVTALRM
: Sygnał ten zostaje wysłany przez funkcję `setitimer()`, gdy upłynął czas dla licznika stworzonego przy użyciu znacznika `ITIMER_VIRTUAL`. Liczniki omówione są w rozdziale 10.

SIGWINCH
: Sygnał ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy rozmiar okna dla ich terminala zostanie zmieniony. Procesy domyślnie ignorują ten sygnał, lecz może zostać on przechwycony i obsłużony, jeśli rozmiar okna terminala jest dla nich przydatnym parametrem. Dobrym przykładem programu, który przechwytuje ten sygnał, jest *top* — wystarczy zmienić rozmiar okna podczas jego działania i obserwować, jak się zachowuje.

SIGXCPU
: Sygnał ten zostaje wysłany przez jądro, gdy proces przekroczy swoje miękkie ograniczenie czasu procesora. Jądro będzie wysyłać ten sygnał co sekundę, dopóki proces nie zakończy swojego działania lub nie przekroczy twardego ograniczenia czasu procesora. Gdy tylko zostanie ono przekroczone, jądro wyśle procesowi sygnał `SIGKILL`.

SIGXFSZ
: Sygnał ten zostaje wysłany przez jądro, gdy proces przekroczy swoje miękkie ograniczenie rozmiaru pliku. Domyślnym zachowaniem jest przerwanie działania procesu, lecz jeśli sygnał ten zostanie przechwycony i zignorowany, wówczas funkcje systemowe, których wywołanie powodowałoby przekroczenie ograniczenia rozmiaru pliku, zwrócą `-1` oraz ustawią zmienną `errno` na `EFBIG`.

Podstawowe zarządzanie sygnałami

Najprostszym i jednocześnie najstarszym interfejsem służącym do zarządzania sygnałami jest funkcja `signal()`. To bardzo prosta funkcja, opisana w standardzie ISO C89 definiującym najmniejszą wspólną część zagadnienia zarządzania sygnałami. Linux oferuje znacznie więcej możliwości kontroli sygnałów dzięki udostępnianiu innych interfejsów, które zostaną omówione w dalszej części rozdziału. Ponieważ funkcja `signal()` jest najbardziej podstawowa, a dzięki obecności w ISO C, również najczęściej używana, dlatego też zostanie zaprezentowana jako pierwsza:

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal (int signo, sighandler_t handler);
```

Poprawne wywołanie funkcji `signal()` instaluje w miejscu aktualnie istniejącej procedury obsługi odebranego sygnału `signo` nową procedurę, wskazywaną przez parametr `handler`. Parametr `signo` jest jedną z nazw sygnałów, takich jak `SIGINT` lub `SIGUSR1`, omówionych w poprzednim podrozdziale. Jak już napisano, proces nie może przechwycić sygnałów `SIGKILL` oraz `SIGSTOP`, dlatego też definiowanie dla nich procedury obsługi nie ma sensu.

Funkcja `handler` musi zwracać `void` — jest to logiczne, ponieważ (w przeciwieństwie do zwykłych funkcji) nie istnieje dla niej standardowe miejsce w programie, do którego mogłaby wracać. `Handler` posiada jeden parametr — liczbę całkowitą, określającą identyfikator sygnału (na przykład `SIGUSR2`), który jest właśnie obsługiwany. Pozwala to na obsługę wielu sygnałów dla pojedynczej funkcji. Prototyp funkcji wygląda następująco:

```
void my_handler (int signo);
```

Linux zamienia go na definicję typu `sighandler_t`. W innych systemach uniksowych używane są bezpośrednio wskaźniki do funkcji, a niektóre systemy definiują własne typy danych, które mogą posiadać inne nazwy niż `sighandler_t`. W programach, dla których ważna jest przenośność, nie powinno się bezpośrednio używać tego typu.

Gdy sygnał zostaje zgłoszony do procesu, który zarejestrował jego procedurę obsługi, wówczas jądro zawiesza wykonywanie zwykłego strumienia instrukcji programu, a zamiast tego wywołuje powyższą procedurę. Jest do niej przekazana wartość informująca o rodzaju sygnału, równa parametrowi `signo`, który pierwotnie został przekazany do funkcji `signal()`.

Funkcji `signal()` można również użyć, aby poinformować jądro, by ignorowało dany sygnał dla aktualnego procesu lub przywróciło dla tego sygnału domyślne zachowanie. Można to uzyskać poprzez użycie specjalnych wartości przekazanych w parametrze `handler`:

`SIG_DFL`
: Dla sygnału `signo` zostanie przywrócone domyślne zachowanie. Na przykład, w przypadku sygnału `SIGPIPE` proces zostanie zakończony.

`SIG_IGN`
: Sygnał `signo` zostanie zignorowany.

Funkcja `signal()` zwraca kod reprezentujący poprzednie zachowanie sygnału, którym może być wskaźnik do procedury obsługi, `SIG_DFL` lub `SIG_IGN`.W przypadku błędu funkcja zwraca `SIG_ERR`, lecz nie ustawia zmiennej `errno`.

Oczekiwanie na dowolny sygnał

Funkcja `pause()`, zdefiniowana w standardzie POSIX i przydatna do celów demonstracyjnych oraz wspomagających uruchamianie programów, pozwala na przełączenie procesu w tryb uśpienia. Proces pozostanie w tym stanie, dopóki nie odbierze sygnału, który zostanie przez niego obsłużony lub spowoduje zakończenie jego działania:

```
#include <unistd.h>

int pause (void);
```

Funkcja `pause()` kończy swoje działanie jedynie wtedy, gdy może przechwycić odebrany sygnał, co oznacza, że zostanie on obsłużony. W tym przypadku funkcja zwraca `-1` i ustawia zmienną `errno` na wartość `EINTR`. Jeśli jądro zgłasza sygnał ignorowany, proces nie zostaje uaktywniony.

Funkcja `pause()` jest jedną z najprostszych funkcji w jądrze Linuksa. Wykonuje tylko dwa działania. Po pierwsze, przełącza proces w tryb uśpienia możliwy do przerwania. Po drugie, wywołuje funkcję `schedule()`, aby uaktywnić zarządcę procesów Linuksa, który powinien znaleźć inny proces do uruchomienia. Ponieważ proces w rzeczywistości na nic nie czeka, nie zostanie uaktywniony przez jądro, dopóki nie odbierze jakiegoś sygnału. Cała czynność zajmuje jedynie dwie linie kodu w języku C[4].

[4] Dlatego też funkcja `pause()` jest drugą pod względem prostoty funkcją systemową w Linuksie. Najprostszymi funkcjami są `getpid()` oraz `getgid()`, z których każda zajmuje tylko jedną linię kodu źródłowego.

Przykłady

Poniżej przedstawione zostaną dwa proste przykłady. Pierwszy z nich rejestruje procedurę obsługi dla sygnału SIGINT, która po prostu wyprowadza wiadomość oraz przerywa działanie programu (co zresztą jest domyślnym zachowaniem tego sygnału):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* procedura obsługi sygnału SIGINT */
static void sigint_handler (int signo)
{
 /*
  * Technicznie rzecz ujmując, funkcja printf() nie powinna być używana
  * w procedurze obsługi sygnału, lecz nie należy
  * się tym przesadnie przejmować. W podrozdziale "Współużywalność"
  * wyjaśnimy, dlaczego tak jest.
  */
    printf ("Przechwycono sygnał SIGINT!\n");
    exit (EXIT_SUCCESS);
}

int main (void)
{
 /*
  * Zarejestruj funkcję sigint_handler jako procedurę
  * obsługi sygnału SIGINT.
  */
    if (signal (SIGINT, sigint_handler) == SIG_ERR)
    {
       fprintf (stderr, "Nie można obsłużyć sygnału SIGINT!\n");
       exit (EXIT_FAILURE);
    }

    for (;;)
       pause ( );

return 0;
}
```

W kolejnym przykładzie przedstawiono zarejestrowanie tej samej procedury dla sygnałów SIGTERM oraz SIGINT. Oprócz tego dla sygnału SIGPROF zostaje przywrócone domyślne zachowanie (którym jest przerwanie działania procesu), natomiast sygnał SIGHUP zostaje zignorowany (choć mógłby w przeciwnym razie również przerwać proces):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

/* procedura obsługi sygnałów SIGINT oraz SIGTERM */
static void signal_handler (int signo)
{
   if (signo == SIGINT)
      printf ("Przechwycono sygnał SIGINT!\n");
   else if (signo == SIGTERM)
      printf ("Przechwycono sygnał SIGTERM!\n");
   else
   {
      /* to nie powinno się nigdy zdarzyć */
```

```
      fprintf (stderr, "Nieoczekiwany sygnał!\n");
      exit (EXIT_FAILURE);
   }

   exit (EXIT_SUCCESS);
}

int main (void)
{
/*
* Zarejestruj funkcję sigint_handler jako procedurę
* obsługi sygnału SIGINT.
*/
if (signal (SIGINT, signal_handler) == SIG_ERR)
{
   fprintf (stderr, "Nie można obsłużyć sygnału SIGINT!\n");
   exit (EXIT_FAILURE);
}

/*
* Zarejestruj funkcję sigint_handler jako procedurę
* obsługi sygnału SIGTERM.
*/
if (signal (SIGTERM, signal_handler) == SIG_ERR)
{
   fprintf (stderr, "Nie można obsłużyć sygnału SIGTERM!\n");
   exit (EXIT_FAILURE);
}

/* Przywróć domyślne zachowanie dla sygnału SIGPROF. */
if (signal (SIGPROF, SIG_DFL) == SIG_ERR)
{
   fprintf (stderr,
    "Nie można przywrócić domyślnego zachowania dla sygnału SIGPROF!\n");
   exit (EXIT_FAILURE);
}

/* Zignoruj sygnał SIGHUP. */
if (signal (SIGHUP, SIG_IGN) == SIG_ERR)
{
   fprintf (stderr, "Nie można zignorować sygnału SIGHUP!\n");
   exit (EXIT_FAILURE);
}

for (;;)
   pause ( );

return 0;
}
```

Uruchomienie i dziedziczenie

Gdy proces zostaje uruchomiony po raz pierwszy, wówczas wszystkim sygnałom zostają przywrócone ich domyślne zachowania, chyba że proces rodzicielski (ten, który uruchomił nowy proces) je ignoruje; w tym przypadku nowo utworzony proces również będzie ignorować te same sygnały. Inaczej mówiąc, w nowo utworzonym procesie będzie przywrócone domyślne zachowanie dla dowolnego sygnału przechwyconego przez proces rodzicielski. Pozostałe sygnały nie zmieniają się. Ma to sens, gdyż nowo uruchomiony proces nie współdzieli przestrzeni adresowej ze swoim rodzicem, dlatego też mogą nie istnieć zarejestrowane dla niego procedury obsługi sygnałów.

To zachowanie, dotyczące uruchamiania procesu, posiada jedno znaczące zastosowanie: gdy powłoka systemowa uruchamia „w tle" jakiś proces (lub gdy proces drugoplanowy uruchamia inny proces), powinien on wówczas ignorować znaki przerwania i wyjścia. W związku z tym zanim powłoka uruchomi proces drugoplanowy, powinna ustawić sygnały `SIGINT` oraz `SIGQUIT` na wartość `SIG_IGN`. Stąd też często spotykanym rozwiązaniem w programach obsługujących powyższe sygnały, jest sprawdzanie, czy nie są one ignorowane. Na przykład:

```
/* obsłuż sygnał SIGINT tylko wtedy, gdy nie jest on ignorowany */
if (signal (SIGINT, SIG_IGN) != SIG_IGN)
{
   if (signal (SIGINT, sigint_handler) == SIG_ERR)
      fprintf (stderr, "Obsługa sygnału SIGINT nie powiodła się!\n");
}

/* obsłuż sygnał SIGQUIT tylko wtedy, gdy nie jest on ignorowany */
if (signal (SIGQUIT, SIG_IGN) != SIG_IGN)
{
   if (signal (SIGQUIT, sigquit_handler) == SIG_ERR)
      fprintf (stderr, "Obsługa sygnału SIGQUIT nie powiodła się!\n");
}
```

Istniejące wymaganie, które nakazuje, aby w celu sprawdzenia sposobu działania sygnału najpierw ustawić jego zachowanie, uwydatnia słabość interfejsu `signal()`. W dalszej części rozdziału przedstawiona zostanie funkcja nieposiadająca takiej wady.

Jak się można spodziewać, sposób działania w przypadku funkcji `fork()`jest odmienny. Gdy proces wywołuje funkcję `fork()`, wówczas potomek dziedziczy dokładnie taką samą semantykę sygnałów, jaką posiada rodzic. Ma to sens, ponieważ potomek i rodzic współdzielą wówczas przestrzeń adresową i dlatego też procedury obsługi sygnałów procesu rodzicielskiego istnieją w procesie potomnym.

Odwzorowanie numerów sygnałów na łańcuchy znakowe

W przedstawionych do tej pory przykładach nazwy sygnałów były na stałe wpisane w kodzie programu. Czasem jednak bardziej wygodną (lub nawet wymaganą) operacją jest przekształcenie numeru sygnału na łańcuch znakowy, reprezentujący jego nazwę. Istnieje kilka metod, aby to uzyskać. Jedną z nich jest odczytywanie łańcucha znaków ze statycznie zdefiniowanej listy:

```
extern const char * const sys_siglist[];
```

Struktura danych `sys_siglist[]` jest tablicą łańcuchów znakowych, przechowującą nazwy sygnałów udostępniane w systemie. Jest indeksowana numerem sygnału.

Alternatywnym rozwiązaniem jest użycie interfejsu `psignal()`, zdefiniowanego w systemie BSD, który jest na tyle popularny, że jest również wspierany przez Linuksa:

```
#include <signal.h>

void psignal (int signo, const char *msg);
```

Wywołanie funkcji `psignal()` wyprowadza na standardowe wyjście błędów `stderr` łańcuch znaków, dostarczony do niej jako parametr `msg`, po którym występuje znak dwukropka, spacja, a wreszcie sama nazwa sygnału z jego numerem podanym w parametrze `signo`. Jeśli parametr `signo` będzie niepoprawny, wówczas informacja o tym znajdzie się w wyprowadzonym tekście.

Lepszym interfejsem jest funkcja `strsignal()`. Nie jest ona zdefiniowana w standardzie, lecz mimo to wspierana przez Linux i wiele innych, nielinuksowych systemów operacyjnych:

```
#define _GNU_SOURCE
#include <string.h>

char *strsignal (int signo);
```

Wywołanie funkcji `strsignal()` zwraca wskaźnik do tekstu, opisującego sygnał reprezentowany przez parametr `signo`. Jeśli wartość `signo` jest niepoprawna, zwrócony opis informuje o tym (w przypadku niektórych systemów uniksowych funkcja zwraca `NULL`). Zwrócony łańcuch znaków pozostaje poprawny jedynie do następnego wywołania funkcji `strsignal()`, gdyż nie obsługuje ona bezpiecznie wątków.

Wykorzystanie tablicy `sys_siglist[]` jest zwykle najlepszym rozwiązaniem. Używając tej metody, można ponownie napisać kod dla wcześniej przedstawionej procedury obsługi sygnałów:

```
static void signal_handler (int signo)
{
   printf ("Przechwycono sygnał %s\n", sys_siglist[signo]);
}
```

Wysyłanie sygnału

Funkcja systemowa `kill()`, będąca podstawą popularnego narzędzia *kill*, wysyła sygnał z jednego procesu do drugiego:

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int signo);
```

W przypadku jej normalnego użycia (gdy wartość `pid` jest większa od zera), funkcja `kill()` wysyła sygnał `signo` do procesu, identyfikowanego przez parametr `pid`.

Jeśli parametr `pid` jest równy zeru, wówczas sygnał `signo` zostanie wysłany do wszystkich procesów z grupy procesów, do której należy proces wywołujący.

Jeśli parametr `pid` jest równy `-1`, wówczas sygnał `signo` zostanie wysłany do procesów, do których procesowi wywołującemu wolno go wysłać, wyłączając samego siebie i proces *init*. Uprawnienia, określające zasady dostarczania sygnałów, zostaną omówione w następnym podrozdziale.

Jeśli parametr `pid` jest mniejszy od `-1`, wówczas sygnał `signo` zostanie wysłany do grupy procesów o identyfikatorze `-pid`.

W przypadku sukcesu funkcja `kill()` zwraca `0`. Wywołanie funkcji kończy się sukcesem, gdy zostanie wysłany przynajmniej jeden sygnał. W przypadku błędu (gdy nie zostanie wysłany żaden sygnał) funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EINVAL`
: Sygnał, reprezentowany przez parametr `signo`, jest nieprawidłowy.

`EPERM`
: Proces wywołujący nie posiada odpowiednich uprawnień, aby wysłać sygnał do dowolnego z żądanych procesów.

ESRCH
: Proces lub grupa procesów, określona w parametrze `pid`, nie istnieje lub (w przypadku procesu) jest procesem zombie.

Uprawnienia

Proces musi posiadać właściwe uprawnienia, aby wysłać sygnał do innego procesu. Proces posiadający uprawnienie `CAP_KILL` (zazwyczaj proces administratora) może wysłać sygnał do dowolnego procesu. Jeśli proces wysyłający nie posiada tego uprawnienia, wówczas jego efektywny lub rzeczywisty identyfikator użytkownika musi być równy rzeczywistemu lub zapisanemu identyfikatorowi użytkownika procesu odbierającego. Podsumowując, użytkownik może wysłać sygnał jedynie do takiego procesu, którego jest właścicielem.

W systemach uniksowych zdefiniowany jest wyjątek `SIGCONT`: proces może wysłać ten sygnał do dowolnego procesu w tej samej sesji. Identyfikatory użytkownika dla tych procesów nie muszą być wówczas sobie równe.

Jeśli wartość `signo` jest równa zeru — wcześniej wspomnianemu sygnałowi zerowemu — wówczas funkcja nie wysyła sygnału, lecz wciąż przeprowadza kontrolę błędów. Jest to użyteczne w przypadku, gdy należy sprawdzić, czy proces posiada odpowiednie uprawnienia, aby wysłać lub przetworzyć sygnał.

Przykłady

Oto, w jaki sposób należy wysłać sygnał `SIGHUP` do procesu o identyfikatorze 1722:

```
int ret;

ret = kill (1722, SIGHUP);
if (ret)
   perror ("kill");
```

Powyższy fragment kodu wykonuje praktycznie taką samą operację jak wywołanie programu użytkowego *kill*:

```
$ kill -HUP 1722
```

Aby sprawdzić, czy istnieją wystarczające uprawnienia pozwalające na wysłanie sygnału do procesu o identyfikatorze 1722, a jednocześnie nie generując podczas tej operacji żadnego sygnału, należy wykonać poniższy kod:

```
int ret;

ret = kill (1722, 0);
if (ret)
   ; /* nie ma uprawnień */
else
   ; /* są uprawnienia */
```

Wysyłanie sygnału do samego siebie

Wykonanie funkcji `raise()` jest prostą metodą pozwalającą na to, aby proces wysłał sygnał do samego siebie:

```
#include <signal.h>

int raise (int signo);
```

Weźmy pod uwagę następujące wywołanie funkcji:

```
raise(signo);
```

Jest ono równe poniższemu kodowi:

```
kill (getpid ( ), signo);
```

W przypadku sukcesu funkcja zwraca wartość `0`, natomiast w przypadku błędu zwraca wartość niezerową. Nie ustawia zmiennej `errno`.

Wysyłanie sygnału do całej grupy procesów

Inna przydatna funkcja pozwala na łatwe wysłanie sygnału do wszystkich procesów z danej grupy, w przypadku, gdy zanegowanie wartości identyfikatora grupy procesów i użycie `kill()` wydaje się być zbyt skomplikowaną operacją:

```
#include <signal.h>

int killpg (int pgrp, int signo);
```

Weźmy pod uwagę następujące wywołanie powyższej funkcji:

```
killpg (pgrp, signo);
```

Jest ono równe kodowi:

```
kill (-pgrp, signo);
```

Powyższe wywołania działają identycznie nawet w przypadku, gdy parametr `pgrp` wynosi zero, co powoduje, że funkcja `killpg()` wysyła sygnał `signo` do grupy procesów, w której znajduje się proces wywołujący.

W przypadku sukcesu funkcja zwraca wartość `0`. W przypadku błędu zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EINVAL`
: Sygnał, przekazany w parametrze `signo`, jest nieprawidłowy.

`EPERM`
: Proces wywołujący nie posiada odpowiednich uprawnień, aby wysłać sygnał do wskazanych procesów.

`ESRCH`
: Grupa procesów, określona w parametrze `pgrp`, nie istnieje.

Współużywalność

Gdy jądro zgłasza sygnał, proces może w tym momencie wykonywać dowolny kod. Na przykład, mógłby znajdować się on w trakcie wykonywania ważnej operacji, która w razie przerwania mogłaby pozostawić go w niespójnym stanie — powiedzmy, struktura danych zostałaby tylko w połowie uaktualniona lub obliczenie wykonałoby się tylko częściowo. Proces mógłby nawet w tym czasie obsługiwać inny sygnał.

Procedury obsługi sygnałów nie posiadają informacji, jaki kod jest wykonywany przez proces, gdy pojawia się sygnał; mogą one zostać uruchomione w trakcie wykonywania dowolnej operacji. Dlatego też jest bardzo ważne, aby dowolna procedura obsługi sygnału, która zostaje zarejestrowana przez proces, przeprowadzała swoje działania i używała zmiennych w rozważny sposób. Podczas tworzenia procedur obsługi sygnałów należy zadbać, aby nie czynić żadnych założeń związanych z tym, w jakim miejscu kodu może znajdować się proces podczas jego przerwania. Szczególna ostrożność jest wskazana podczas modyfikowania zmiennych globalnych (to znaczy współdzielonych). Zasadniczo dobrym pomysłem jest założenie, że procedura obsługi zdarzeń nie może modyfikować takich zmiennych. W kolejnym podrozdziale omówiony zostanie sposób, pozwalający na tymczasowe zablokowanie mechanizmu dostarczania sygnałów, będący rozwiązaniem, by bezpiecznie modyfikować dane współdzielone przez procedurę obsługi oraz pozostałą część procesu.

W jaki sposób powyżej wspominany problem związany jest z funkcjami systemowymi i innymi funkcjami bibliotecznymi? Co się stanie, gdy proces jest w trakcie zapisywania do pliku lub przydzielania pamięci, a procedura obsługi sygnału zacznie zapisywać do tego samego pliku lub wywoła funkcję `malloc()`? Jaki będzie skutek, gdy proces jest w trakcie wykonywania funkcji, takiej jak `strsignal()`, która używa bufora statycznego, i nastąpi dostarczenie sygnału?

Niektóre funkcje nie są w oczywisty sposób współużywalne. Jeśli program jest w trakcie wykonywania funkcji, która nie jest współużywalna, a zostanie wygenerowany sygnał i procedura jego obsługi rozpocznie wykonywanie tej samej funkcji, wówczas może wkraść się chaos. *Funkcja współużywalna* jest funkcją, którą można bezpiecznie wywołać z niej samej (lub współbieżnie z innego wątku tego samego procesu). Aby funkcja była współużywalna, nie może modyfikować danych statycznych, może natomiast modyfikować jedynie takie dane, które zostały przydzielone na stosie lub dostarczone przez program wywołujący. Funkcja ta nie może również wywoływać żadnych innych funkcji, które nie są współużywalne.

Funkcje, dla których współużywalność jest zagwarantowana

Podczas tworzenia procedury obsługi sygnału należy założyć, że przerwany proces może być w trakcie wykonywania funkcji, która nie jest współużywalna (lub w trakcie wykonywania innego fragmentu kodu). Dlatego też procedury obsługi sygnałów mogą używać jedynie takich funkcji, które są współużywalne.

W wielu standardach zdefiniowano listy funkcji bezpiecznych dla sygnałów, czyli współużywalnych i nadających się do zastosowania w procedurach obsługi. Najbardziej znane z nich, POSIX.1-2003 oraz Single UNIX Specification, definiują listy funkcji, dla których współużywalność jest zagwarantowana we wszystkich zgodnych platformach. W tabeli 9.2. przedstawiono listę tych funkcji.

Istnieje dużo więcej funkcji, które są bezpieczne, lecz w Linuksie oraz innych systemach, zgodnych ze standardem POSIX, współużywalność jest gwarantowana tylko dla powyżej przedstawionych.

Tabela 9.2. Funkcje, które mogą być bezpiecznie współużywalne podczas obsługi sygnałów

abort()	accept()	access()
aio_error()	aio_return()	aio_suspend()
alarm()	bind()	cfgetispeed()
cfgetospeed()	cfsetispeed()	cfsetospeed()
chdir()	chmod()	chown()
clock_gettime()	close()	connect()
creat()	dup()	dup2()
execle()	execve()	Exit()
_exit()	fchmod()	fchown()
fcntl()	fdatasync()	fork()
fpathconf()	fstat()	fsync()
ftruncate()	getegid()	geteuid()
getgid()	getgroups()	getpeername()
getpgrp()	getpid()	getppid()
getsockname()	getsockopt()	getuid()
kill()	link()	listen()
lseek()	lstat()	mkdir()
mkfifo()	open()	pathconf()
pause()	pipe()	poll()
posix_trace_event()	pselect()	raise()
read()	readlink()	recv()
recvfrom()	recvmsg()	rename()
rmdir()	select()	sem_post()
send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()
setsockopt()	setuid()	shutdown()
sigaction()	sigaddset()	sigdelset()
sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()
sigprocmask()	sigqueue()	sigset()
sigsuspend()	sleep()	socket()
socketpair()	stat()	symlink()
Sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()
time()	timer_getoverrun()	timer_gettime()
timer_settime()	times()	umask()
uname()	unlink()	utime()
wait()	waitpid()	write()

Zbiory sygnałów

Niektóre funkcje, przedstawione w dalszej części rozdziału, operują na zbiorach sygnałów, takich jak grupy sygnałów blokowanych przez proces lub grupy sygnałów oczekujących na obsłużenie. Takie zbiory sygnałów mogą być przetwarzane za pomocą *operacji zbioru sygnałów*:

```
#include <signal.h>

int sigemptyset (sigset_t *set);

int sigfillset (sigset_t *set);

int sigaddset (sigset_t *set, int signo);

int sigdelset (sigset_t *set, int signo);

int sigismember (const sigset_t *set, int signo);
```

Funkcja `sigemptyset()` inicjalizuje zbiór sygnałów, podany w parametrze `set`, oznaczając go jako pusty (wszystkie sygnały są wykluczone ze zbioru). Funkcja `sigfillset()` inicjalizuje zbiór sygnałów, podany w parametrze `set`, oznaczając go jako pełny (wszystkie sygnały należą do zbioru). Obie funkcje zwracają zero. Powinny być wywołane dla danego zbioru sygnałów przed jego późniejszym użyciem.

Funkcja `sigaddset()` dodaje sygnał `signo` do zbioru sygnałów przekazanego w parametrze `set`, natomiast funkcja `sigdelset()` usuwa z niego sygnał `signo`. Obie funkcje zwracają zero w przypadku sukcesu, natomiast `−1` w przypadku niepowodzenia, podczas którego ustawiają zmienną `errno` na wartość kodu błędu `EINVAL`, oznaczającego, że `signo` jest niepoprawnym identyfikatorem sygnału.

Funkcja `sigismember()` zwraca `1`, jeśli `signo` znajduje się w zbiorze sygnałów podanym w parametrze `set`. Zwraca `0`, jeśli `signo` nie znajduje się w tym zbiorze lub `−1` w przypadku błędu, podczas którego zmienna `errno` zostaje ponownie ustawiona na wartość `EINVAL`, oznaczającą, że identyfikator `signo` jest niepoprawny.

Inne funkcje obsługujące zbiory sygnałów

Poprzednio przedstawione funkcje są zdefiniowane w standardzie POSIX i można je znaleźć w każdym nowoczesnym systemie uniksowym. Linux udostępnia również kilka niestandardowych funkcji:

```
#define _GNU_SOURCE
#define <signal.h>

int sigisemptyset (sigset_t *set);

int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);

int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);
```

Funkcja `sigisemptyset()` zwraca `1`, gdy zbiór sygnałów, reprezentowany przez parametr `set`, jest pusty. W przeciwnym przypadku zwraca `0`.

Funkcja `sigorset()` umieszcza unię (sumę binarną) zbiorów sygnałów `left` i `right` w zbiorze `dest`. Funkcja `sigandset()` umieszcza przecięcie (iloczyn binarny) zbiorów sygnałów `left`

i `right` w zbiorze `dest`. Obie funkcje zwracają `0` w przypadku sukcesu, a `−1` w przypadku niepowodzenia i ustawiają wówczas zmienną `errno` na wartość `EINVAL`.

Funkcje te są użyteczne, lecz nie powinno się ich używać w programach, które wymagają pełnej zgodności ze standardem POSIX.

Blokowanie sygnałów

We wcześniejszej części tego rozdziału omówiono mechanizm współużywalności oraz problemy, jakie mogą powstawać podczas asynchronicznego działania procedur obsługi sygnałów. Pokazano, że funkcje, które nie są współużywalne, nie mogą być wywoływane z procedury obsługi sygnału.

Co dzieje się w sytuacji, gdy program wymaga współdzielenia danych między procedurą obsługi sygnału a jakimś innym dowolnym fragmentem kodu? Jak wygląda sytuacja w przypadku, gdy istnieją pewne obszary wykonania programu, które nie mogą zostać niczym przerwane, wliczając w to przerwanie spowodowane przez procedury obsługi sygnałów? Takie miejsca w programie zwane są *rejonami krytycznymi* i mogą być one chronione poprzez tymczasowe wstrzymanie dostarczania sygnałów. Sygnały takie to sygnały *zablokowane*. Dowolny sygnał, który zostaje zgłoszony podczas tego stanu, nie będzie obsłużony, dopóki sygnały nie zostaną ponownie odblokowane. Proces może blokować dowolną liczbę sygnałów; zbiory sygnałów blokowanych przez proces zwane są jego *maską sygnałową*.

W standardzie POSIX zdefiniowano, a w systemie Linux zaimplementowano funkcję, która zarządza maską sygnałową procesu:

```
#include <signal.h>

int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);
```

Zachowanie funkcji `sigprocmask()` zależy od wartości parametru `how`, który może być równy jednemu z poniżej podanych znaczników:

`SIG_SETMASK`
: Maska sygnałowa procesu wywołującego została zmieniona na wartość `set`.

`SIG_BLOCK`
: Sygnały, zdefiniowane w zbiorze sygnałów `set`, zostaną dodane do maski sygnałowej procesu wywołującego. Finalna maska sygnałowa będzie wynikiem operacji unii (binarnej sumy) aktualnej wartości tej maski oraz zbioru `set`.

`SIG_UNBLOCK`
: Sygnały, zdefiniowane w zbiorze sygnałów `set`, zostaną usunięte z maski sygnałowej procesu wywołującego. Finalna maska sygnałowa będzie wynikiem operacji przecięcia (binarnego iloczynu) aktualnej wartości tej maski oraz negacji (binarnego odwrócenia bitów) zbioru `set`.

Jeśli parametr `oldset` nie jest równy `NULL`, wówczas funkcja umieszcza w nim poprzednią wartość zbioru sygnałów.

Jeśli parametr `set` jest równy `NULL`, wówczas funkcja ignoruje parametr `how` i nie zmienia maski sygnałowej, lecz w dalszym ciągu umieszcza ją w `oldset`. Przekazanie wartości zerowej w parametrze `set` jest sposobem na odczytanie aktualnej maski sygnałowej.

W przypadku sukcesu, funkcja zwraca wartość 0. W przypadku błędu, zwraca −1 oraz odpowiednio ustawia zmienną errno na wartość EINVAL, oznaczającą, że parametr how jest niepoprawny, lub na wartość EFAULT, wskazującą, że parametry set lub oldset są błędnymi wskaźnikami.

Blokowanie sygnałów SIGKILL bądź SIGSTOP nie jest dozwolone. Funkcja sigprocmask() w sposób przezroczysty ignoruje próby ich dodania do maski sygnałowej.

Odzyskiwanie oczekujących sygnałów

Gdy jądro zgłasza sygnał blokowany, nie zostaje on dostarczony. Sygnały takie zwane są *oczekującymi*. Gdy sygnał oczekujący będzie odblokowany, jądro przekaże go do procesu w celu obsłużenia.

POSIX definiuje funkcję, która odzyskuje zbiór oczekujących sygnałów:

```
#include <signal.h>

int sigpending (sigset_t *set);
```

Poprawne wywołanie funkcji sigpending() umieszcza zbiór oczekujących sygnałów w parametrze set oraz zwraca 0. W przypadku błędu, funkcja zwraca −1 oraz ustawia zmienną errno na wartość EFAULT, oznaczającą, że set jest niepoprawnym wskaźnikiem.

Oczekiwanie na zbiór sygnałów

Trzecia funkcja, zdefiniowana w standardzie POSIX, pozwala procesowi na chwilową zmianę jego maski sygnałowej, a następnie oczekiwanie, dopóki nie zostanie zgłoszony sygnał, który przerwie działanie tego procesu lub zostanie przez niego obsłużony:

```
#include <signal.h>

int sigsuspend (const sigset_t *set);
```

Jeśli sygnał przerywa działanie procesu, wykonanie funkcji sigsuspend() nie zakończy się. Kiedy sygnał zostanie zgłoszony i obsłużony, wówczas funkcja sigsuspend() zwróci wartość −1, gdy procedura obsługi sygnału się zakończy, jednocześnie ustawiając zmienną errno na wartość EINTR. Jeśli set będzie niepoprawnym wskaźnikiem, wtedy zmienna errno zostanie ustawiona na EFAULT.

Standardowym scenariuszem użycia funkcji sigsuspend() jest odzyskiwanie sygnałów, które mogły nadejść i zostać zablokowane podczas wykonywania sekcji krytycznej w trakcie działania programu. Proces najpierw używa funkcji sigprocmask(), aby zablokować zbiór sygnałów, zapisując poprzednią maskę do parametru oldset. Po wyjściu z rejonu krytycznego proces wywołuje sigsuspend(), przekazując wartość oldset do parametru set.

Zaawansowane zarządzanie sygnałami

Funkcja signal(), która została omówiona na początku tego rozdziału, jest bardzo prosta. Ponieważ jest częścią standardowej biblioteki języka C, dlatego też musi odpowiadać minimalnym wymaganiom dotyczącym możliwości sytemu operacyjnego, w którym działa. Powoduje to, że zakres jej działania ograniczony jest jedynie do obsługi najmniejszej wspólnej części zagad-

nienia zarządzania sygnałami. Jako alternatywę dla niej, standard POSIX definiuje funkcję systemową `sigaction()`, która udostępnia dużo więcej możliwości zarządzania sygnałami. Między innymi można użyć jej do blokowania odbioru niektórych sygnałów, gdy procedura obsługi jest wciąż aktywna, a także do uzyskiwania szerokiego zakresu informacji dotyczących systemu oraz stanu procesu w momencie, gdy został zgłoszony sygnał:

```
#include <signal.h>

int sigaction (int signo, const struct sigaction *act, struct sigaction *oldact);
```

Wywołanie funkcji `sigaction()` zmienia zachowanie sygnału identyfikowanego przez parametr `signo`, który może posiadać dowolną wartość, za wyjątkiem `SIGKILL` oraz `SIGSTOP`. Jeśli parametr `act` nie jest równy `NULL`, wówczas funkcja systemowa zmienia aktualne zachowanie sygnału, który zostaje w nim przekazany. Jeśli parametr `oldact` nie jest równy `NULL`, wówczas funkcja przechowuje w nim poprzednie (lub aktualne, jeśli wartość `act` jest równa `NULL`) zachowanie dla danego sygnału.

Struktura `sigaction` pozwala na dokładną kontrolę zachowania sygnałów. Jest zdefiniowana w pliku nagłówkowym `<sys/signal.h>`, dołączonym do pliku `<signal.h>`:

```
struct sigaction
{
   void (*sa_handler)(int); /* procedura obsługi sygnału lub rodzaj akcji */
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask; /* sygnały do zablokowania */
   int sa_flags; /* znaczniki */
   void (*sa_restorer)(void); /* pole przestarzałe i niekompatybilne z POSIX */
}
```

Pole `sa_handler` określa rodzaj akcji, która musi zostać podjęta po otrzymaniu sygnału. Podobnie jak w przypadku funkcji `signal()` pole to może być równe `SIG_DFL`, co oznacza akcję domyślną, `SIG_IGN`, informujące w tym przypadku jądro, aby ignorował sygnał dla procesu lub może być również wskaźnikiem do funkcji obsługi sygnału. Funkcja ta posiada taki sam prototyp jak procedura obsługi sygnału instalowana przez `signal()`:

```
void my_handler (int signo);
```

Jeśli pole `sa_flags` będzie równe znacznikowi `SA_SIGINFO`, wówczas definicję funkcji obsługi sygnału określać będzie pole `sa_sigaction`, a nie `sa_handler`. Prototyp tej funkcji jest trochę inny:

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

Funkcja otrzymuje numer sygnału w swoim pierwszym parametrze, a strukturę `siginfo_t` w drugim, natomiast struktura `ucontext_t` (zrzutowana jako wskaźnik do `void`) zostaje przekazana w trzecim parametrze. Funkcja nie posiada wartości powrotnej. Struktura `siginfo_t` dostarcza dużo informacji do procedury obsługi sygnału; zostanie ona wkrótce omówiona.

Należy zauważyć, że w przypadku niektórych architektur maszynowych (i prawdopodobnie innych systemów uniksowych) pola `sa_handler` oraz `sa_sigaction` są unią i nie należy przypisywać im jednocześnie wartości.

Pole `sa_mask` dostarcza zestawu sygnałów, które powinny być zablokowane przez system w czasie wykonywania procedury obsługi. Pozwala to programistom na uzyskiwanie prawidłowej ochrony przed współużywalnością pomiędzy różnymi procedurami obsługi sygnałów.

Aktualnie obsługiwany sygnał będzie zablokowany, dopóki w polu `sa_flags` nie pojawi się znacznik `SA_NODEFER`. Nie można blokować sygnałów `SIGKILL` oraz `SIGSTOP`; funkcja w sposób niewidoczny będzie je ignorować w polu `sa_mask`.

Pole `sa_flags` jest maską bitową zera, jednego lub większej liczby znaczników, które zmieniają obsługę sygnału podanego w parametrze `signo`. Do tej pory omówiono już dwa znaczniki: `SA_SIGINFO` oraz `SA_NODEFER`; inne wartości pola `sa_flags` składają się również z poniższych znaczników:

`SA_NOCLDSTOP`
: Jeśli parametr `signo` jest równy `SIGCHLD`, wówczas znacznik ten informuje system, aby nie dostarczał powiadomienia, gdy proces potomny zostaje zatrzymany lub ponownie uruchomiony.

`SA_NOCLDWAIT`
: Jeśli parametr `signo` jest równy `SIGCHLD`, wówczas znacznik ten uaktywnia *automatyczne przechwytywanie potomków*: procesy potomne nie zostają zamienione na procesy zombie po zakończeniu ich działania, a proces rodzicielski nie musi (nie może) wywoływać dla nich funkcji `wait()`. Szczegółowe informacje dotyczące procesów potomnych, zombie i funkcji `wait()` znajdują się w rozdziale 5.

`SA_NOMASK`
: Znacznik ten jest przestarzałym, niezdefiniowanym przez POSIX równoważnikiem znacznika `SA_NODEFER` (omówionego już w tym podrozdziale). Zamiast tego znacznika można podać `SA_NODEFER`, lecz należy być przygotowanym na to, że w starszym kodzie zamiast niego pojawi się `SA_NOMASK`.

`SA_ONESHOT`
: Znacznik ten jest przestarzałym, niezdefiniowanym przez POSIX równoważnikiem znacznika `SA_RESETHAND` (który zostanie wkrótce omówiony). Zamiast tego znacznika można podać `SA_RESETHAND`, lecz należy być przygotowanym na to, że w starszym kodzie zamiast niego pojawi się `SA_ONESHOT`.

`SA_ONSTACK`
: Znacznik ten informuje system, aby wywołał daną procedurę obsługi, używając w tym celu *alternatywnego stosu sygnałowego* udostępnionego przez funkcję `signalstack()`. Jeśli alternatywny stos nie zostanie dostarczony, będzie użyty domyślny — oznacza to, że system zachowa się tak, jak gdyby nie użyto znacznika `SA_ONSTACK`. Alternatywne stosy sygnałowe stosuje się rzadko, chociaż są one użyteczne dla pewnych aplikacji *pthreads* posiadających stos o mniejszym rozmiarze, który może zostać przepełniony podczas użycia procedur obsługi sygnałów.

`SA_RESTART`
: Znacznik ten uaktywnia proces ponownego uruchamiania funkcji systemowych, przerwanych przez sygnały, który jest charakterystyczny dla systemu BSD.

`SA_RESETHAND`
: Znacznik ten uaktywnia tryb „jednorazowego działania”. Gdy tylko procedura obsługi zakończy swoje działanie, wówczas zachowaniu danego sygnału zostaną przywrócone wartości domyślne.

Pole `sa_restorer` jest przestarzałe i nie jest już używane w systemie Linux. Nie jest również zdefiniowane w standardzie POSIX. Należy przyjąć, że pole to po prostu nie istnieje i w ogóle go nie używać.

Funkcja `sigaction()` zwraca `0` w przypadku sukcesu. W przypadku błędu zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EFAULT`
: Parametry `act` lub `oldact` są niepoprawnymi wskaźnikami.

`EINVAL`
: Parametr `signo` jest niepoprawnym sygnałem lub jest równy wartościom `SIGKILL` albo `SIGSTOP`.

Struktura siginfo_t

Struktura `siginfo_t` jest również zdefiniowana w pliku nagłówkowym `<sys/signal.h>` i wygląda następująco:

```
typedef struct siginfo_t
{
   int si_signo;       /* numer sygnału */
   int si_errno;       /* wartość błędu */
   int si_code;        /* kod sygnału */
   pid_t si_pid;       /* PID dla procesu wysyłającego */
   uid_t si_uid;       /* rzeczywisty UID dla procesu wysyłającego */
   int si_status;      /* kod wyjścia lub sygnał */
   clock_t si_utime;   /* zużyty czas użytkownika */
   clock_t si_stime;   /* zużyty czas systemowy */
   sigval_t si_value;  /* wartość pola użytkowego dla sygnału */
   int si_int;         /* sygnał POSIX.1b */
   void *si_ptr;       /* sygnał POSIX.1b */
   void *si_addr;      /* obszar pamięci, który spowodował błąd */
   int si_band;        /* zdarzenie poza kolejką */
   int si_fd;          /* deskryptor pliku */
};
```

Struktura ta zawiera dużo informacji, które zostają przekazane do procedury obsługi sygnału (jeśli zamiast pola `sa_sighandler` zostanie użyte pole `sa_sigaction`). Uwzględniając stan nowoczesnej techniki komputerowej, uważa się, że model sygnałów systemu Unix nie pozwala na prostą realizację komunikacji międzyprocesowej (IPC). Być może problemem jest to, że niepotrzebnie używa się funkcji `signal()`, kiedy należałoby stosować raczej funkcję `sigaction()`, uruchamianą z ustawionym znacznikiem `SA_SIGINFO`. Struktura `sigaction_t` pozwala na uzyskanie większej funkcjonalności podczas użycia sygnałów.

Znajduje się w niej wiele użytecznych danych, włącznie z informacjami dotyczącymi procesu, który wysłał sygnał wraz z powodem jego wygenerowania. Oto dokładny opis każdego z pól struktury:

`si_signo`
: Pole to określa numer danego sygnału. Tę samą informację dostarcza pierwszy parametr procedury obsługi sygnału (jego użycie zapobiega odwoływaniu się do wartości dostępnej przez wskaźnik, co ma miejsce w przypadku użycia struktury `siginfo_t`).

`si_errno`
: Jeśli wartość ta jest różna od zera, oznacza kod błędu związany z sygnałem. Pole to jest poprawne dla wszystkich sygnałów.

`si_code`
: Zawiera wyjaśnienie, dlaczego i skąd proces otrzymał dany sygnał (na przykład z funkcji `kill()`). Dopuszczalne wartości tego pola zostaną omówione w następnym podrozdziale. Jest ono poprawne dla wszystkich sygnałów.

`si_pid`
: W przypadku sygnału `SIGCHLD` pole to określa identyfikator procesu, który został przerwany.

`si_uid`
: W przypadku sygnału `SIGCHLD` pole to określa właścicielski identyfikator użytkownika dla procesu, który został przerwany.

`si_status`
: W przypadku sygnału `SIGCHLD` pole to określa kod wyjścia dla procesu, który został przerwany.

`si_utime`
: W przypadku sygnału `SIGCHLD` pole to określa czas użytkownika zużyty przez proces, który został przerwany.

`si_stime`
: W przypadku sygnału `SIGCHLD` pole to określa czas systemowy zużyty przez proces, który został przerwany.

`si_value`
: Unia pól `si_int` oraz `si_ptr`.

`si_int`
: Dla sygnałów wysłanych za pomocą funkcji `sigqueue()` (szczegóły w podrozdziale Wysyłanie sygnału z wykorzystaniem pola użytkowego, znajdującym się w dalszej części rozdziału) pole to oznacza dostarczoną wartość pola użytkowego, przedstawioną w postaci liczby całkowitej.

`si_ptr`
: Dla sygnałów wysłanych za pomocą funkcji `sigqueue()` (szczegóły w podrozdziale Wysyłanie sygnału z wykorzystaniem pola użytkowego, znajdującym się w dalszej części rozdziału) pole to oznacza dostarczoną wartość pola użytkowego, przedstawioną w postaci wskaźnika `void`.

`si_addr`

W przypadku sygnałów `SIGBUS`, `SIGFPE`, `SIGILL`, `SIGSEGV` oraz `SIGTRAP` ten wskaźnik `void` zawiera adres powstania błędu. Na przykład, w przypadku `SIGSEGV` pole zawiera adres, w którym została naruszona ochrona pamięci (stąd też często jest on równy `NULL`!).

`si_band`

W przypadku sygnału `SIGPOLL` pole to zawiera informacje priorytetowe oraz dane poza kolejką dla deskryptora pliku, podanego w polu `si_fd`.

`si_fd`

W przypadku sygnału `SIGPOLL` pole to jest deskryptorem pliku, dla którego zostały zakończone operacje.

Pola `si_value`, `si_int` oraz `si_ptr` dotyczą złożonej problematyki, ponieważ mogą zostać użyte przez dany proces, aby przesłać dowolne dane do innego procesu. Dlatego też można ich użyć, aby wysłać zarówno zwykłą liczbę całkowitą, jak również wskaźnik do struktury danych (należy zwrócić uwagę na to, że przesłanie wskaźnika nie przyda się, jeśli procesy nie współdzielą przestrzeni adresowej między sobą). Pola te zostaną omówione w dalszym podrozdziale Wysyłanie sygnału z wykorzystaniem pola użytkowego.

POSIX zapewnia, że tylko pierwsze trzy pola będą poprawne dla wszystkich sygnałów. Inne pola powinny być używane jedynie wówczas, gdy należy obsłużyć odpowiedni sygnał. Na przykład, pola `si_fd` należy używać jedynie wówczas, gdy obsługiwanym sygnałem jest `SIGPOLL`.

Wspaniały świat pola si_code

Pole `si_code` opisuje przyczynę wygenerowania sygnału. Dla sygnałów wysyłanych przez użytkownika pole informuje, w jaki sposób zostały one wysłane. Dla sygnałów wysyłanych przez jądro, wskazuje, dlaczego zostały one wysłane.

Poniżej przedstawione zostaną możliwe wartości pola `si_code`, które są poprawne dla wszystkich sygnałów. Oznaczają, w jaki sposób lub z jakiego powodu został wysłany dany sygnał:

`SI_ASYNCIO`
: Sygnał został wysłany z powodu zakończenia asynchronicznych operacji wejścia i wyjścia (szczegóły w rozdziale 5.).

`SI_KERNEL`
: Sygnał został zgłoszony przez jądro.

`SI_MESGQ`
: Sygnał został wysłany z powodu zmiany stanu kolejki wiadomości POSIX (nieuwzględnionej w tej książce).

`SI_QUEUE`
: Sygnał został wysłany przez funkcję `sigqueue()` (opis w następnym podrozdziale).

`SI_TIMER`
: Sygnał został wysłany w wyniku upłynięcia czasu dla licznika POSIX (szczegóły w rozdziale 10.).

`SI_TKILL`
: Sygnał został wysłany przez funkcje `tkill()` lub `tgkill()`. Są one używane w bibliotekach wątkowych i nie zostały uwzględnione w niniejszej publikacji.

`SI_SIGIO`
: Sygnał został wysłany z powodu umieszczenia sygnału `SIGIO` w kolejce.

`SI_USER`
: Sygnał został wysłany przez funkcje `kill()` lub `raise()`.

Poniżej zostaną przedstawione możliwe wartości pola `si_code`, poprawne jedynie dla sygnału `SIGBUS`. Oznaczają one rodzaj błędu sprzętowego, który wystąpił:

`BUS_ADRALN`
: Proces spowodował powstanie błędu wyrównania (szczegółowe informacje na temat wyrównania znajdują się w rozdziale 8.).

`BUS_ADRERR`
: Proces próbował uzyskać dostęp do niepoprawnego adresu fizycznego.

`BUS_OBJERR`
: Proces spowodował powstanie innego rodzaju błędu sprzętowego.

W przypadku sygnału SIGCHLD następujące wartości opisują przyczynę wysłania sygnału od procesu potomnego do rodzica:

CLD_CONTINUED
: Proces potomny był zatrzymany, lecz został ponownie uruchomiony.

CLD_DUMPED
: Proces potomny zakończył się w niepoprawny sposób.

CLD_EXITED
: Proces potomny zakończył się w poprawny sposób poprzez wykonanie funkcji exit().

CLD_KILLED
: Proces potomny został usunięty.

CLD_STOPPED
: Proces potomny został zatrzymany.

CLD_TRAPPED
: Proces potomny natrafił na pułapkę.

Poniżej zostaną przedstawione wartości pola si_code, poprawne jedynie dla sygnału SIGPFE. Opisują rodzaj błędu arytmetycznego, który wystąpił:

FPE_FLTDIV
: Proces wykonał operację zmiennoprzecinkową, która zakończyła się dzieleniem przez zero.

FPE_FLTOVF
: Proces wykonał operację zmiennoprzecinkową, która zakończyła się przepełnieniem.

FPE_FLTINV
: Proces wykonał błędną operację zmiennoprzecinkową.

FPE_FLTRES
: Proces wykonał operację zmiennoprzecinkową, która spowodowała powstanie niedokładnego lub błędnego wyniku.

FPE_FLTSUB
: Proces wykonał operację zmiennoprzecinkową, która używała indeksu spoza zakresu.

FPE_FLTUND
: Proces wykonał operację zmiennoprzecinkową, która zakończyła się niedomiarem.

FPE_INTDIV
: Proces wykonał operację całkowitoliczbową, która zakończyła się dzieleniem przez zero.

FPE_INTOVF
: Proces wykonał operację całkowitoliczbową, która zakończyła się przepełnieniem.

Poniżej zostaną przedstawione możliwe wartości pola si_code, które są poprawne jedynie dla sygnału SIGILL. Opisują przyczynę wykonania niepoprawnej instrukcji:

ILL_ILLADR
: Proces zamierzał użyć niepoprawnego trybu adresowania.

ILL_ILLOPC
: Proces zamierzał wykonać niepoprawny kod operacji.

ILL_ILLOPN
: Proces zamierzał użyć niepoprawnego argumentu.

`ILL_PRVOPC`
: Proces zamierzał wykonać uprzywilejowany kod operacji.

`ILL_PRVREG`
: Proces miał być wykonany w uprzywilejowanym rejestrze.

`ILL_ILLTRP`
: Proces zamierzał wkroczyć do niepoprawnej pułapki.

W przypadku wszystkich powyższych wartości pole `si_addr` zawiera adres powstania błędu.

Poniżej zostaną przedstawione możliwe wartości pola `si_code`, które są poprawne dla sygnału `SIGPOLL`. Opisują zdarzenie związane z operacjami wejścia i wyjścia, które spowodowało wygenerowanie sygnału:

`POLL_ERR`
: Wystąpił błąd operacji wejścia i wyjścia.

`POLL_HUP`
: Urządzenie zawiesiło się lub gniazdo zostało odłączone.

`POLL_IN`
: Plik posiada dane dostępne do odczytu.

`POLL_MSG`
: Wiadomość jest dostępna.

`POLL_OUT`
: Można wykonać operację zapisu dla pliku.

`POLL_PRI`
: Plik posiada dane o wysokim poziomie priorytetu, które mogą zostać odczytane.

Poniżej zostaną przedstawione możliwe wartości pola `si_code`, które są poprawne dla sygnału `SIGSEGV`. Opisują dwa rodzaje niewłaściwego dostępu do pamięci:

`SEGV_ACCERR`
: Proces próbował w nieprawidłowy sposób uzyskać dostęp do poprawnego obszaru w pamięci — oznacza to, że naruszył uprawnienia dostępu do pamięci.

`SEGV_MAPERR`
: Proces próbował uzyskać dostęp do niepoprawnego obszaru pamięci.

W przypadku powyższych dwóch wartości pole `si_addr` zawiera adres powstania błędu.

Poniżej zostaną przedstawione dwie możliwe wartości pola `si_code`, które są poprawne dla sygnału `SIGTRAP`. Opisują rodzaj pułapki:

`TRAP_BRKPT`
: Proces natrafił na punkt wstrzymania.

`TRAP_TRACE`
: Proces natrafił na pułapkę śledzenia.

Należy zwrócić uwagę na to, że `si_code` jest polem zawierającym wartość, a nie polem bitowym.

Wysyłanie sygnału z wykorzystaniem pola użytkowego

Jak zostało pokazane w poprzednim podrozdziale, procedury obsługi sygnałów, zarejestrowane przy użyciu znacznika SA_SIGINFO, otrzymują parametr o typie siginfo_t. Struktura ta zawiera element zwany si_value, który jest opcjonalnym polem użytkowym, przekazywanym między źródłem sygnału a jego odbiorcą.

Funkcja sigqueue(), zdefiniowana w standardzie POSIX, pozwala procesowi na wysłanie sygnału z wykorzystaniem tego pola użytkowego:

```
#include <signal.h>

int sigqueue (pid_t pid, int signo, const union sigval value);
```

Funkcja sigqueue() działa podobnie jak funkcja kill(). W przypadku sukcesu sygnał, który identyfikowany jest przez parametr signo, zostaje umieszczony w kolejce dla procesu lub grupy procesów reprezentowanej przez pid. Następnie funkcja zwraca 0. Pole użytkowe dla sygnału reprezentowane jest przez parametr value, który jest unią liczby całkowitej oraz wskaźnika void:

```
union sigval
{
   int sival_int;
   void *sival_ptr;
};
```

W przypadku błędu funkcja zwraca −1 oraz odpowiednio ustawia zmienną errno na jedną z poniższych wartości:

EINVAL
: Sygnał, reprezentowany przez parametr signo, jest nieprawidłowy.

EPERM
: Proces wywołujący nie posiada odpowiednich uprawnień, aby wysłać sygnał do dowolnego żądanego procesu. Uprawnienia, niezbędne dla wysłania sygnału, są takie same jak w przypadku funkcji kill() (szczegóły w podrozdziale Wysyłanie sygnału, znajdującym się we wcześniejszej części rozdziału).

ESRCH
: Proces lub grupa procesów reprezentowana przez parametr pid nie istnieje lub (w przypadku procesu) jest procesem zombie.

Podobnie jak ma to miejsce w przypadku funkcji kill(), w parametrze signo można przekazać wartość sygnału zerowego (0), aby przetestować uprawnienia.

Przykład

Poniższy przykład wysyła do procesu o identyfikatorze 1722 sygnał SIGUSR2, który zawiera pole użytkowe o typie całkowitoliczbowym, równym wartości 404:

```
sigval value;
int ret;

value.sival_int = 404;
```

```
ret = sigqueue (1722, SIGUSR2, value);
if (ret)
   perror ("sigqueue");
```

Jeśli proces o identyfikatorze 1722 obsłuży sygnał `SIGUSR2` za pomocą procedury używającej znacznika `SA_SIGINFO`, wówczas parametr `signo` zostanie ustawiony na `SIGUSR2`, pole `si->si_int` będzie posiadać wartość `404`, natomiast pole `si->si_code` będzie równe `SI_QUEUE`.

Zakończenie

Sygnały posiadają złą opinię wśród wielu programistów Uniksa. Są starym rozwiązaniem, nieodpowiednim dla zaimplementowania wymiany informacji między jądrem a użytkownikiem oraz realizują — w najlepszym przypadku — prymitywną formę komunikacji międzyprocesowej (IPC).

Mimo tych wad są mniej lub bardziej potrzebne. Sygnały to jedyne narzędzia umożliwiające odbieranie wielu powiadomień (takich jak powiadomień o wykonaniu niepoprawnego kodu operacji) z jądra. Za pomocą sygnałów system Unix (a także Linux) przerywa działanie procesów i zarządza relacjami między rodzicami i potomkami. Dlatego też muszą być one używane.

Jednym z podstawowych powodów unikania użycia sygnałów jest to, że nie istnieje prosty sposób na napisanie poprawnej procedury obsługi, która bezpiecznie obsługuje sytuacje związane ze współużywalnością. Jeśli jednak procedury obsługi sygnałów będą proste i użyte zostaną w nich jedynie takie funkcje, których listę zawiera tabela 9.2. (jeśli w ogóle jakieś będą użyte!), wówczas powinny działać bezpiecznie.

Inną, słabą stroną sygnałów jest fakt, że wielu programistów wciąż używa funkcji `signal()` i `kill()`, zamiast stosować `sigaction()` oraz `sigqueue()`, by nimi zarządzać. Jak pokazano w ostatnich dwóch podrozdziałach, sygnały udostępniają dużo więcej możliwości oraz informacji, gdy użyte zostaną dla nich procedury obsługi wykorzystujące znacznik `SA_SIGINFO`. Pomimo że lepszym rozwiązaniem byłoby całkowite odejście od sygnałów na rzecz odpytywanego mechanizmu wykorzystującego deskryptory plików, który jest rozpatrywany dla przyszłych wersji jądra Linuksa, jednak unikanie niepewnych metod oraz wykorzystywanie zaawansowanych interfejsów obsługi sygnałów pozwala na złagodzenie niedogodności związanych z ich użyciem.

ROZDZIAŁ 10.

Czas

W nowoczesnych systemach operacyjnych czas wykorzystywany jest w wielu miejscach. W wielu programach wymagane jest jego śledzenie. Jądro odmierza upływ czasu za pomocą trzech sposobów:

Czas rzeczywisty (ang. *real time* lub *wall time*)
: Jest to faktyczny czas, istniejący w świecie rzeczywistym, to znaczy taki, który mógłby zostać odczytany z zegara zawieszonego na ścianie. Procesy używają czasu rzeczywistego podczas współpracy z użytkownikiem lub zapamiętywania momentu wystąpienia zdarzenia.

Czas procesu (ang. *process time*)
: Jest to czas, który został zużyty zarówno przez proces podczas bezpośredniego wykonywania kodu z przestrzeni użytkownika, jak również pośrednio poprzez jądro, które go obsługiwało. Ten rodzaj odmierzania upływu czasu ważny jest dla procesów przede wszystkim ze względu na operacje profilowania i uzyskiwania danych statystycznych — na przykład, aby ustalić, jak długo wykonywała się dana czynność. W tych przypadkach użycie czasu rzeczywistego może powodować powstanie błędów, ponieważ ze względu na wielozadaniowy charakter samego systemu Linux, czas procesu dla danej operacji może być dużo mniejszy od czasu rzeczywistego. W trakcie oczekiwania na operacje wejścia i wyjścia (szczególnie podczas obsługi wprowadzania danych z klawiatury) proces może zużyć znaczącą liczbę cykli maszynowych.

Czas monotoniczny (ang. *monotonic time*)
: W przypadku tej opcji czas narasta liniowo. Większość systemów operacyjnych, włączając w to również Linuksa, używa systemowego licznika czasu bezawaryjnej pracy (czasu, który upłynął od momentu załadowania systemu). Czas rzeczywisty może ulec zmianie — na przykład, zostanie on zmieniony przez użytkownika lub przez system, gdy konieczne będzie przeprowadzenie operacji dopasowania czasu ze względu na pojawiające się odchylenia. Mogą jeszcze pojawić się dodatkowe nieścisłości, jak na przykład sekundy przestępne. Z drugiej strony, systemowy czas bezawaryjnej pracy jest deterministyczny i niezmienny. Ważnym aspektem w przypadku źródła czasu monotonicznego nie jest jego aktualna wartość, lecz gwarancja, że czas ten będzie narastać liniowo, co pozwoli na obliczanie różnic pomiędzy dwoma zadanymi momentami.

Dlatego też czas monotoniczny jest odpowiedni dla obliczania czasu względnego, podczas gdy czas rzeczywisty stosuje się do pomiarów czasu bezwzględnego.

Trzy przedstawione powyżej sposoby pomiaru czasu mogą być reprezentowane w postaci jednej lub dwóch form:

Czas względny
: Jest to wartość względna w stosunku do pewnej wartości wzorcowej, takiej jak aktualna chwila: na przykład, *5 sekund od tego momentu*, *10 minut temu*.

Czas absolutny
: To czas bez uwzględnienia jakichkolwiek wartości wzorcowych, na przykład: *południe, 25 marca 1968*.

Wspomniane formy czasu posiadają swoje zastosowanie. Proces mógłby zdecydować się na przerwanie danego żądania za 500 milisekund, odświeżać zawartość ekranu 60 razy na sekundę lub otrzymać informację, że od rozpoczęcia operacji minęło 7 sekund. Wszystkie te przykłady opisują zastosowanie obliczeń wykorzystujących czas względny. I przeciwnie, aplikacja kalendarzowa może zapamiętać datę przyjęcia towarzyskiego, planowanego na 8 lutego, system plików zapisuje pełną datę i czas utworzenia pliku (zamiast informacji „pięć sekund temu"), a zegar użytkownika wyświetla datę w formacie kalendarza gregoriańskiego, a nie liczbę sekund, które upłynęły od załadowania systemu operacyjnego.

Systemy uniksowe reprezentują czas absolutny jako liczbę sekund, które upłynęły od daty początku epoki. Początek epoki zdefiniowany został na dzień 1 stycznia 1970 roku, godzinę 00:00:00 uniwersalnego czasu koordynowanego (UTC). UTC (ang. *Universal Time Coordinated*) jest w przybliżeniu równy czasowi GMT (ang. *Greenwich Mean Time*) oraz Zulu. Oznacza to, że w systemie Unix nawet czas absolutny jest czasem względnym, postrzeganym z niskopoziomowego punktu widzenia. Unix udostępnia specjalny typ danych służący przechowywaniu „sekund, które minęły od początku epoki". Zostanie on omówiony w następnym podrozdziale.

Śledzenie postępu czasu w systemach operacyjnych możliwe jest poprzez użycie zegara programowego, który zarządzany jest przez oprogramowanie jądra. Jądro tworzy egzemplarz licznika okresowego, znanego pod nazwą *zegara systemowego* (ang. *system timer*), który sterowany jest zegarem o odpowiedniej częstotliwości. Gdy przedział czasowy licznika kończy się, wówczas jądro zwiększa zużyty czas o jedną jednostkę, nazywaną *taktem* (ang. *tick*) lub *chwilą* (ang. *jiffy*). Licznik zużytych taktów zwany jest *licznikiem chwil* (ang. *jiffies counter*). Licznik ten dla wersji 2.6 jądra Linuksa jest 64-bitowy, w porównaniu z poprzednią, 32-bitową wartością[1].

W systemie Linux częstotliwość zegara systemowego reprezentowana jest przez parametr `HZ`, ponieważ odpowiada mu definicja preprocesora o tej samej nazwie. Wartość parametru `HZ` zależy od architektury i nie jest częścią interfejsu binarnego aplikacji Linuksa — oznacza to, że programy nie mogą być zależne od tej wartości oraz nie można oczekiwać, że będzie ona równa pewnej konkretnej liczbie. Dawniej w architekturze x86 używana była wartość `100`, oznaczająca, że zegar systemowy generował 100 taktów na sekundę (posiadał częstotliwość 100 herców). Stąd czas każdego taktu równy był wartości 0,01 sekundy — zgodnie ze wzorem `1/HZ`. Wraz z udostępnieniem wersji 2.6 jądra Linuksa, jego projektanci zwiększyli wartość `HZ` do `1000`, ustalając jednocześnie czas każdego taktu na 0,001 sekundy. Jednak dla wersji 2.6.13 i później-

[1] Przyszłe wersje jądra Linuksa mogą być „beztaktowe" lub może zostać dla nich zaimplementowane „taktowanie dynamiczne", polegające na tym, że jądro nie będzie bezpośrednio śledzić liczby zużytych taktów. Wszystkie wykorzystujące czas operacje jądra będą używały dynamicznie tworzonych liczników, zamiast korzystać z zegara systemowego.

szych, HZ wynosi 250, a czas taktu równy jest 0,004 sekundy [2]. Określona wartość HZ powoduje pewne skutki: wyższe wartości pozwalają na uzyskanie lepszej rozdzielczości, lecz jednocześnie powodują większe obciążenie zegara.

Mimo że procesy nie powinny polegać na żadnej ustalonej wartości parametru HZ, standard POSIX definiuje mechanizm, który pozwala określić częstotliwość zegara systemowego w trakcie działania programu:

```
long hz;

hz = sysconf (_SC_CLK_TCK);
if (hz == -1)
   perror ("sysconf"); /* ten błąd nie powinien nigdy wystąpić */
```

Interfejs ten jest przydatny, gdy w programie należy ustalić rozdzielczość zegara systemowego, lecz nie jest wymagany, aby zamienić wartości czasu systemowego na sekundy, ponieważ większość interfejsów standardu POSIX udostępnia rezultaty pomiaru czasu, które są już odpowiednio przekształcone lub dopasowane do pewnej ustalonej częstotliwości, niezależnej od wartości HZ. W przeciwieństwie do HZ, ta ustalona częstotliwość jest częścią systemowego interfejsu binarnego aplikacji (ABI); w przypadku architektury x86 wynosi 100. Funkcje standardu POSIX, które zwracają czas w postaci taktów zegarowych, używają stałej CLOCKS_PER_SEC reprezentującej wspomnianą częstotliwość ustaloną.

Pojawienie się jakiegoś zdarzenia powoduje od czasu do czasu konieczność wyłączenia komputera. Zdarza się, że komputery są całkowicie wyłączone nawet przez dłuższy czas. Po załadowaniu systemu udostępniają jednak poprawną wartość czasu. Dzieje się tak, ponieważ większość komputerów posiada *zegar sprzętowy*, zasilany bateryjnie, który przechowuje wartości czasu i daty, podczas gdy komputer jest wyłączony. W trakcie swojego ładowania do systemu jądro inicjalizuje parametry aktualnego czasu danymi pobranymi z zegara sprzętowego. Administrator systemu może synchronizować czas w dowolnym momencie poprzez użycie komendy *hwclock*.

Zarządzanie upływem czasu w systemie Unix pociąga za sobą wykonywanie pewnych czynności, z których tylko kilka dotyczy danego procesu: należą do nich ustawianie i odczytywanie aktualnego czasu rzeczywistego, obliczanie upływu czasu, zatrzymywanie działania przez określony czas, wykonywanie pomiarów czasu o wysokiej dokładności, sterowanie licznikami. W rozdziale tym zostaną poddane analizie wszystkie zagadnienia związane z czasem. Na początku zaprezentujemy struktury danych, które używane są w systemie Linux do reprezentowania czasu.

Struktury danych reprezentujące czas

W trakcie ewolucji systemu Unix, podczas której zaimplementowano w nim interfejsy służące do obsługi czasu, pojawiło się wiele struktur danych, które mogły reprezentować na pozór prostą koncepcję czasu. Obejmują one szeroki zakres typów danych, poczynając od prostych liczb całkowitych, a kończąc na strukturach o wielu polach. Analizie zostaną poddane wspomniane struktury danych, a następnie właściwe interfejsy.

[2] Parametr HZ jest również opcją jądra, ustalaną w czasie kompilacji, która może być równa jednej z trzech możliwych wartości: 100, 250 lub 1000. Wartości te są udostępnione dla architektury x86. Mimo tego programy z przestrzeni użytkownika nie mogą być zależne od żadnej konkretnej wartości parametru HZ.

Reprezentacja pierwotna

Najprostszą strukturą danych jest `time_t`, która zdefiniowana została w pliku nagłówkowym `<time.h>`. W założeniach miała być typem nieprzejrzystym. Jednak w wielu systemach uniksowych (włącznie z Linuksem), typ ten jest prostym zamiennikiem nazwy dla typu `long` języka C, uzyskanym za pomocą polecenia `typedef`:

```
typedef long time_t;
```

Typ `time_t` reprezentuje liczbę sekund, które upłynęły od początku epoki. Typową uwagą dotyczącą przedstawionej definicji jest zdanie: „To nie wystarczy na długo!". W rzeczywistości zliczanie sekund potrwa dłużej, niż można się tego spodziewać, lecz faktycznie licznik przepełni się, gdy wciąż będą używane liczne systemy uniksowe. Przy zastosowaniu 32-bitowego typu `long` struktura `time_t` może zapamiętać maksymalnie 2 147 483 676 sekund, które upłynęły od początku epoki. Oznacza to, że problem roku 2000 pojawi się ponownie w roku *2038*! Mając jednak trochę szczęścia, można oczekiwać, że w poniedziałek, 18 stycznia 2038, o godzinie 22:14:07 większość systemów i oprogramowania będzie działało na platformach 64-bitowych.

Następna wersja — dokładność na poziomie mikrosekund

Innym problemem związanym z typem `time_t`, jest to, że w ciągu jednej sekundy może się wiele wydarzyć. Struktura `timeval` rozszerza typ `time_t` poprzez dodanie dokładności mikrosekundowej i jest zdefiniowana w pliku nagłówkowym `<sys/time.h>`:

```
#include <sys/time.h>

struct timeval
{
   time_t tv_sec; /* sekundy */
   suseconds_t tv_usec; /* mikrosekundy */
};
```

Pole `tv_sec` zlicza sekundy, natomiast pole `tv_usec` zlicza mikrosekundy. Wprowadzający w błąd typ `suseconds_t` jest zazwyczaj zwykłym zamiennikiem nazwy (`typedef`) dla typu całkowitoliczbowego.

Kolejna, lepsza wersja — dokładność na poziomie nanosekund

Dokładność do mikrosekund nie była wystarczająca, więc „podniesiono stawkę" i dla struktury `timespec` ustalono nanosekundową dokładność czasu. Struktura ta jest zdefiniowana w pliku nagłówkowym `<time.h>`:

```
#include <time.h>

struct timespec
{
   time_t tv_sec; /* sekundy */
   long tv_nsec; /* nanosekundy */
};
```

Mając możliwość wyboru, w interfejsach preferowane jest użycie dokładności do nanosekund, a nie do mikrosekund[3]. Zgodnie z tym od momentu wprowadzenia struktury `timespec`, zaim-

[3] Dodatkowo implementacja struktury `timespec` pozwala na uniknięcie użycia niezbyt przemyślanego typu `suseconds_t`, a zamiast niego zastosowanie prostego i bezpretensjonalnego typu `long`.

plementowano jej użycie w większości interfejsów, które obsługiwały zagadnienia związane z czasem i uzyskano lepszą dokładność. Jednak, jak zostanie pokazane, jedna ważna funkcja wciąż używa struktury `timeval`.

W praktyce użycie dowolnej z tych struktur nie pozwala na uzyskanie żądanej precyzji, ponieważ zegar systemowy nie udostępnia zarówno nanosekundowej, jak i nawet mikrosekundowej rozdzielczości. Mimo to zaleca się, aby używać dokładności dostępnej w interfejsie, gdyż może zostać ona dopasowana do dowolnej rozdzielczości oferowanej przez system.

Wyłuskiwanie składników czasu

Niektóre omówione funkcje umożliwiają zamianę czasu reprezentowanego w Uniksie na odpowiednie łańcuchy znakowe lub pozwalają na programowe tworzenie łańcuchów odpowiadających danej dacie. Aby ułatwić realizację tych zadań, w standardzie języka C udostępniono strukturę `tm`, która reprezentuje „rozłożone" składniki czasu przedstawione w bardziej czytelny sposób. Struktura ta jest również zdefiniowana w pliku nagłówkowym `<time.h>`:

```
#include <time.h>
struct tm
{
   int tm_sec;             /* sekundy */
   int tm_min;             /* minuty */
   int tm_hour;            /* godziny */
   int tm_mday;            /* dzień miesiąca */
   int tm_mon;             /* miesiąc */
   int tm_year;            /* rok */
   int tm_wday;            /* dzień tygodnia */
   int tm_yday;            /* dzień roku */
   int tm_isdst;           /* czas letni? */
#ifdef _BSD_SOURCE
   long tm_gmtoff;       /* przesunięcie strefy czasowej w stosunku do GMT */
   const char *tm_zone; /* łańcuch znaków, zawierający skrót strefy czasowej */
#endif /* _BSD_SOURCE */
};
```

Użycie struktury `tm` ułatwia uzyskanie informacji, czy wartość `time_t`, równa na przykład `314159`, odpowiada niedzieli bądź sobocie (faktycznie jest niedzielą). Z punktu widzenia zajętej pamięci wybór tej struktury do przechowywania daty i czasu nie jest rozsądną decyzją, lecz pozwala na łatwą konwersję do wartości przyjaznych dla użytkownika.

Poszczególne pola struktury `tm` są następujące:

`tm_sec`
: Liczba sekund po pełnej minucie. Wartość ta zwykle zawiera się w zakresie od `0` do `59`, lecz może być także równa nawet `61`, co oznacza w tym przypadku dwie sekundy przestępne.

`tm_min`
: Liczba minut po pełnej godzinie. Wartość ta zawiera się w zakresie od `0` do `59`.

`tm_hour`
: Liczba godzin po północy. Wartość ta zawiera się w zakresie od `0` do `23`.

`tm_mday`
: Dzień miesiąca. Wartość ta zawiera się w zakresie od `0` do `31`. Standard POSIX nie definiuje wartości `0`; w Linuksie jest ona jednak używana, aby określić ostatni dzień poprzedniego miesiąca.

`tm_mon`
: Liczba miesięcy od stycznia. Wartość ta zawiera się w zakresie od `0` do `11`.

`tm_year`
: Liczba lat od roku 1900.

`tm_wday`
: Liczba dni od niedzieli. Wartość ta zawiera się w zakresie od `0` do `6`.

`tm_yday`
: Liczba dni od pierwszego stycznia. Wartość ta zawiera się w zakresie od `0` do `365`.

`tm_isdst`
: Wartość specjalna, informująca, czy dla czasu zdefiniowanego w innych polach istnieje aktywna opcja czasu letniego. Jeśli wartość ta jest dodatnia, czas letni jest aktywny. Jeśli wartość wynosi `0`, czas letni nie jest zastosowany. Jeśli wartość jest ujemna, stan opcji czasu letniego jest nieznany.

`tm_gmtoff`
: Przesunięcie (w sekundach) aktualnej strefy czasowej w stosunku do GMT. Pole to istnieje jedynie wówczas, gdy przed instrukcją, dołączającą plik `<time.h>`, zdefiniowano stałą `_BSD_SOURCE`.

`tm_zone`
: Skrót nazwy aktualnej strefy czasowej, na przykład CET. Pole to istnieje jedynie wówczas, gdy przed instrukcją, dołączającą plik `<time.h>`, zdefiniowano stałą `_BSD_SOURCE`.

Typ danych dla czasu procesu

Typ danych `clock_t` reprezentuje takty zegara. Jest to typ całkowitoliczbowy, często równy typowi `long`. W zależności od interfejsu takty w `clock_t` oznaczają rzeczywistą częstotliwość zegara systemowego (`HZ`) lub wartość `CLOCKS_PER_SEC`.

Zegary POSIX

Niektóre omówione w tym rozdziale funkcje systemowe używają *zegarów POSIX* (ang. *POSIX clocks*) definiujących standard, pozwalający na zaimplementowanie i reprezentowanie źródeł czasu. Typ danych `clockid_t` reprezentuje określony zegar POSIX, którego cztery rodzaje są wspierane przez system Linux:

`CLOCK_MONOTONIC`
: Zegar z monotonicznie rosnącą wartością czasu, którego nie można ustawiać za pomocą żadnego procesu. Wskazuje czas, który upłynął od pewnego nieznanego wcześniej punktu początkowego, takiego jak ładowanie systemu.

`CLOCK_PROCESS_CPUTIME_ID`
: Zegar o wysokiej rozdzielczości, dostępny dla każdego procesu z osobna, udostępniony przez procesor. Na przykład, w architekturze i386 zegar ten używa rejestru licznika sygnałów czasowych (TSC).

`CLOCK_REALTIME`
: Ogólnosystemowy zegar czasu rzeczywistego. Aby go ustawić, należy posiadać specjalne uprawnienia.

CLOCK_THREAD_CPUTIME_ID
: Podobny do zegara dostępnego dla wszystkich procesów, lecz unikalny dla każdego z ich wątków.

POSIX definiuje wszystkie cztery powyżej opisane źródła czasu, lecz wymaga istnienia jedynie CLOCK_REALTIME. Dlatego też, mimo że Linux w sposób solidny wspiera wszystkie powyżej przedstawione rodzaje zegarów, kod przenośny powinien być oparty wyłącznie na CLOCK_
↪REALTIME.

Rozdzielczość źródła czasu

Standard POSIX definiuje funkcję clock_getres() pozwalającą na odczytanie wartości rozdzielczości danego źródła czasu:

```
#include <time.h>

int clock_getres (clockid_t clock_id, struct timespec *res);
```

Poprawne wywołanie funkcji clock_getres() zapisuje w strukturze, wskazywanej przez wskaźnik res (w przypadku, gdy jest on różny od NULL), rozdzielczość zegara podanego w parametrze clock_id oraz zwraca 0. W przypadku błędu funkcja zwraca −1 oraz odpowiednio ustawia zmienną errno na jedną z dwóch poniższych wartości:

EFAULT
: Parametr res jest niepoprawnym wskaźnikiem.

EINVAL
: Parametr clock_id nie jest poprawnym źródłem czasu w tym systemie.

Poniższy przykład wyprowadza wartości rozdzielczości czterech źródeł czasu omówionych w poprzednim podrozdziale:

```
clockid_t clocks[] =
{
   CLOCK_REALTIME,
   CLOCK_MONOTONIC,
   CLOCK_PROCESS_CPUTIME_ID,
   CLOCK_THREAD_CPUTIME_ID,
   (clockid_t) -1
};
int i;

for (i = 0; clocks[i] != (clockid_t) -1; i++)
{
   struct timespec res;
   int ret;

   ret = clock_getres (clocks[i], &res);
   if (ret)
      perror ("clock_getres");
   else
      printf ("zegar=%d sekundy=%ld nanosekundy=%ld\n", clocks[i], res.tv_sec,
   res.tv_nsec);
}
```

W nowoczesnym systemie x86 otrzymane wyniki są podobne do poniżej przedstawionych:

```
zegar=0 sekundy=0 nanosekundy=4000250
zegar=1 sekundy=0 nanosekundy=4000250
```

```
zegar=2 sekundy=0 nanosekundy=1
zegar=3 sekundy=0 nanosekundy=1
```

Należy zwrócić uwagę na to, że 4 000 250 nanosekund jest równe czterem milisekundom, co z kolei równa się 0,004 sekundy. Z kolei 0,004 sekundy jest rozdzielczością zegara systemowego działającego w architekturze x86, co daje wartość `HZ` równą `250` — jest to zgodne z tym, co podano w pierwszym podrozdziale niniejszego rozdziału. Dlatego też oba zegary `CLOCK_REALTIME` oraz `CLOCK_MONOTONIC` powiązane są z taktowaniem, natomiast rozdzielczość zależy od zegara systemowego. I odwrotnie, oba zegary `CLOCK_PROCESS_CPUTIME_ID` oraz `CLOCK_THREAD_CPUTIME_ID` używają źródeł czasu o wysokiej rozdzielczości — dla maszyny x86 jest to rejestr licznika sygnałów czasowych (TSC), który umożliwia uzyskanie dokładności na poziomie nanosekund.

W przypadku Linuksa (i większości innych systemów uniksowych) dla wszystkich funkcji, które używają zegarów POSIX, wymagane jest łączenie wynikowego obiektu z biblioteką *librt*. Na przykład, jeśli zaistniałaby potrzeba skompilowania poprzednio przedstawionego fragmentu kodu do pełnego pliku wykonywalnego, należałoby użyć następującego polecenia:

```
$ gcc -Wall -W -O2 –lrt -g -o snippet snippet.c
```

Pobieranie aktualnego czasu

W aplikacjach z różnych powodów wymaga się dostępu do aktualnej wartości czasu i daty: aby przekazać te informacje użytkownikowi, by obliczyć czas względny lub miniony, zapamiętać czas wystąpienia zdarzenia itd. Najprostszym i historycznie najbardziej rozpowszechnionym sposobem otrzymywania wartości aktualnego czasu jest użycie funkcji `time()`:

```
#include <time.h>

time_t time (time_t *t);
```

Wywołanie funkcji `time()` zwraca aktualny czas reprezentowany przez liczbę sekund, które upłynęły od początku epoki. Jeśli parametr `t` nie jest równy `NULL`, wówczas funkcja również zapisuje aktualną wartość czasu do dostarczonego wskaźnika.

W przypadku błędu funkcja zwraca wartość `–1` (zrzutowaną na typ `time_t`) oraz odpowiednio ustawia zmienną `errno`. Jedynym możliwym kodem błędu jest `EFAULT`, oznaczający, że `t` jest niepoprawnym wskaźnikiem.

Na przykład

```
time_t t;

printf ("aktualny czas: %ld\n", (long) time (&t));
printf ("ta sama wartość: %ld\n", (long) t);
```

Lepszy interfejs

Funkcja `gettimeofday()` jest rozszerzoną wersją `time()` udostępniającą dokładność na poziomie mikrosekund:

```
#include <sys/time.h>

int gettimeofday (struct timeval *tv, struct timezone *tz);
```

Proste podejście do zagadnienia czasu

Reprezentacja czasu, zrealizowana przez zastosowanie typu `time_t` i oznaczająca „sekundy, które minęły od początku epoki", nie odzwierciedla rzeczywistej liczby sekund, które upłynęły od tego pamiętnego momentu. W systemie Unix założono, że lata przestępne to wszystkie te podzielne przez cztery, natomiast całkowicie zignorowano sekundy przestępne. Oznacza to, że reprezentacja czasu za pomocą typu danych `time_t` nie jest dokładna, lecz została zrealizowana w konsekwentny sposób. I tak w rzeczywistości jest.

Poprawne wywołanie funkcji `gettimeofday()` umieszcza wartość aktualnego czasu w strukturze `timeval` wskazywanej przez parametr `tv` oraz zwraca `0`. Struktura `timezone` oraz parametr `tz` są przestarzałe; nie powinny być używane w Linuksie. W parametrze `tz` należy zawsze przekazywać wartość `NULL`.

W przypadku błędu funkcja zwraca `-1` oraz ustawia zmienną `errno` na wartość `EFAULT`; jest to jedyny możliwy kod błędu, który oznacza, że parametry `tv` lub `tz` są niepoprawnymi wskaźnikami.

Na przykład:

```
struct timeval tv;
int ret;

ret = gettimeofday (&tv, NULL);
if (ret)
   perror ("gettimeofday");
else
   printf ("sekudny=%ld mikrosekundy=%ld\n", (long) tv.sec, (long) tv.usec);
```

Struktura `timezone` jest przestarzała, ponieważ jądro nie zarządza strefami czasowymi, a biblioteka *glibc* nie pozwala na użycie pola `tz_dsttime` pochodzącego z tej struktury. W dalszym podrozdziale przedstawiony zostanie sposób pozwalający na modyfikowanie strefy czasowej.

Interfejs zaawansowany

Standard POSIX udostępnia interfejs `clock_gettime()`, który pozwala na otrzymywanie czasu z określonego źródła. Bardziej użyteczną cechą tej funkcji jest jednak to, że pozwala ona na uzyskanie dokładności na poziomie nanosekund:

```
#include <time.h>

int clock_gettime (clockid_t clock_id, struct timespec *ts);
```

W przypadku sukcesu, funkcja zwraca wartość zero oraz zapisuje odpowiednią wartość aktualnego czasu w strukturze `ts` dla źródła czasu określonego w parametrze `clock_td`. W przypadku błędu, zwraca `-1` i odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EFAULT`
: Parametr `ts` jest niepoprawnym wskaźnikiem.

`EINVAL`
: Parametr `clock_id` reprezentuje niepoprawne źródło czasu w danym systemie.

Następujący przykład pozwala na otrzymanie wartości aktualnego czasu ze wszystkich czterech standardowych źródeł czasu:

```
clockid_t clocks[] =
{
   CLOCK_REALTIME,
   CLOCK_MONOTONIC,
   CLOCK_PROCESS_CPUTIME_ID,
   CLOCK_THREAD_CPUTIME_ID,
   (clockid_t) -1
};
int i;

for (i = 0; clocks[i] != (clockid_t) -1; i++)
{
   struct timespec ts;
   int ret;

   ret = clock_gettime (clocks[i], &ts);
   if (ret)
      perror ("clock_gettime");
   else
      printf ("zegar=%d sekundy=%ld nanosekundy=%ld\n", clocks[i], ts.tv_sec,
  ts.tv_nsec);
}
```

Pobieranie czasu procesu

Funkcja systemowa `times()` pozwala na odczytanie czasu dla działającego procesu i jego potomków, o wielkości reprezentowanej w taktach zegarowych:

```
#include <sys/times.h>

struct tms
{
   clock_t tms_utime;  /* wykorzystany czas użytkownika */
   clock_t tms_stime;  /* wykorzystany czas systemowy */
   clock_t tms_cutime; /* czas użytkownika wykorzystany przez procesy potomne */
   clock_t tms_cstime; /* czas systemowy wykorzystany przez procesy potomne */
};

clock_t times (struct tms *buf);
```

W przypadku sukcesu, funkcja wypełnia odpowiednie pola struktury `tms` wskazywanej przez parametr wskaźnikowy `buf`. W strukturze zapisywany jest czas procesu zużyty przez proces wywołujący i jego potomków. Zwracane wartości podzielone są na czas użytkownika i czas systemowy. *Czas użytkownika* to czas zużyty podczas wykonywania kodu w przestrzeni użytkownika. *Czas systemowy* to czas zużyty podczas wykonywania kodu w przestrzeni jądra — na przykład wywołania funkcji systemowej lub obsługi błędu stronicowania. Zwracane wartości czasów dla każdego potomka dodawane są w momencie, gdy proces potomny przestaje działać, a rodzic wywołuje dla jego obsługi funkcję `waitpid()` (lub inną funkcję pokrewną). Funkcja zwraca liczbę taktów zegarowych, przyrastających od pewnego momentu w sposób monotoniczny. Tym momentem było kiedyś załadowanie systemu operacyjnego, dlatego też funkcja `times()` zwracała czas działania systemu (w taktach zegarowych). Obecnie punkt odniesienia wyprzedza o około 429 milionów sekund moment uruchomienia systemu operacyjnego. Modyfikacja ta została zaimplementowana przez programistów jądra, by skorygować jego kod, który nie mógł poprawnie obsługiwać pomiaru czasu działania systemu w przypadku przepełnienia

licznika i rozpoczęcia zliczania od zera. Wartość bezwzględna, zwracana przez tę funkcję, jest nieprzydatna, ale znajduje zastosowanie podczas określania względnych różnic czasowych między jej dwoma wywołaniami.

W przypadku błędu funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno`. Dla Linuksa jedynym możliwym kodem błędu jest `EFAULT`, który oznacza, że parametr `buf` jest niepoprawnym wskaźnikiem.

Ustawianie aktualnego czasu

W poprzednich podrozdziałach przedstawiono metody pozwalające na odczytanie czasu. W aplikacjach istnieje także konieczność ustawienia aktualnego czasu i daty na wartość, która zostaje dostarczona. Jest to prawie zawsze wykonywane za pomocą narzędzi zaprojektowanych wyłącznie do tego celu, takich jak program o nazwie *date*.

Odpowiednikiem funkcji `time()`, ustawiającym czas, jest funkcja `stime()`:

```
#define _SVID_SOURCE
#include <time.h>

int stime (time_t *t);
```

Poprawne wywołanie funkcji `stime()` ustawia czas systemowy na wartość wskazywaną przez parametr `t` oraz zwraca `0`. Aby funkcja wykonała się poprawnie, wywołujący ją użytkownik musi posiadać uprawnienie `CAP_SYS_TIME`. Zazwyczaj posiada je tylko użytkownik administracyjny.

W przypadku błędu funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na wartość `EFAULT`, oznaczającą, że parametr `t` jest niepoprawnym wskaźnikiem lub na wartość `EPERM`, która wskazuje, że użytkownik wywołujący nie posiadał uprawnienia `CAP_SYS_TIME`.

Użycie funkcji jest bardzo proste:

```
time_t t = 1;
int ret;

/* ustaw czas na jedną sekundę po rozpoczęciu epoki */
ret = stime (&t);
if (ret)
   perror ("stime");
```

W dalszym podrozdziale zostaną omówione funkcje, które w prostszy sposób przekształcają czytelne formy, reprezentujące wartość czasu, na typ danych `time_t`.

Precyzyjne ustawianie czasu

Odpowiednikiem funkcji `gettimeofday()` jest funkcja `settimeofday()`:

```
#include <sys/time.h>

int settimeofday (const struct timeval *tv, const struct timezone *tz);
```

Poprawne wywołanie funkcji `settimeofday()` ustawia czas systemowy na wartość podaną w parametrze `tv` oraz zwraca `0`. Podobnie jak w przypadku funkcji `gettimeofday()`, w parametrze `tz` należy przekazać wartość `NULL`. W przypadku błędu funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

EFAULT
: Parametry `tv` lub `tz` wskazują na niepoprawny obszar pamięci.

EINVAL
: W jednej z dostarczonych struktur znajduje się niepoprawne pole.

EPERM
: Proces wywołujący nie posiada uprawnienia `CAP_SYS_TIME`.

Poniższy przykład ustawia aktualny czas na wartość równą sobocie, gdzieś w środku grudnia 1979 roku:

```
struct timeval tv =
   { .tv_sec = 31415926,
     .tv_usec = 27182818 };
int ret;

ret = settimeofday (&tv, NULL);
if (ret)
   perror ("settimeofday");
```

Zaawansowany interfejs ustawiania czasu

Podobnie jak funkcja `clock_gettime()` jest ulepszoną wersją funkcji `gettimeofday()`, tak funkcja `clock_settime()` jest następcą funkcji `settimeofday()`:

```
#include <time.h>

int clock_settime (clockid_t clock_id, const struct timespec *ts);
```

W przypadku sukcesu funkcja zwraca `0`, a dla źródła czasu określonego w parametrze `clock_id`, zostaje ustawiony czas przekazany w parametrze `ts`. W przypadku błędu funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

EFAULT
: Parametr `ts` jest niepoprawnym wskaźnikiem.

EINVAL
: Parametr `clock_id` reprezentuje niepoprawne źródło czasu w danym systemie.

EPERM
: Proces nie posiada wymaganych uprawnień pozwalających na ustawianie określonego źródła czasu lub dane źródło czasu nie może być ustawione.

W większości systemów jedynym źródłem, które może zostać ustawione, jest `CLOCK_REALTIME`. Dlatego też jedyną przewagą tej funkcji nad `settimeofday()` jest to, że oferuje ona dokładność na poziomie nanosekund (równocześnie nie zmuszając użytkownika do korzystania z bezwartościowej struktury `timezone`).

Konwersje czasu

Systemy uniksowe oraz język C dostarczają zestawu funkcji pozwalających na konwersję pomiędzy składnikami czasu rozłożonego (łańcuchem znakowym ASCII, reprezentującym czas), a typem `time_t`. Funkcja `asctime()` przekształca strukturę `tm` (czas rozłożony) na łańcuch znakowy ASCII:

```
#include <time.h>

char * asctime (const struct tm *tm);
char * asctime_r (const struct tm *tm, char *buf);
```

Funkcja zwraca wskaźnik do statycznie przydzielonego łańcucha znakowego. Kolejne wywołanie dowolnej funkcji przetwarzającej czas może nadpisać ten łańcuch; funkcja `asctime()` nie obsługuje wątków w bezpieczny sposób.

Dlatego też programy wielowątkowe (oraz programiści, którzy nie przepadają za niewłaściwie zaprojektowanymi interfejsami) powinny używać funkcji `asctime_r()`. Zamiast zwracać wskaźnik do statycznie przydzielonego łańcucha znakowego, funkcja ta używa łańcucha przekazanego w parametrze `buf`, który powinien mieć długość co najmniej 26 znaków.

Obie funkcje zwracają `NULL` w przypadku błędu.

Funkcja `mktime()` również zamienia strukturę `tm`, lecz w tym przypadku na typ danych `time_t`:

```
#include <time.h>

time_t mktime (struct tm *tm);
```

Funkcja `mktime()` ustawia także strefę czasową poprzez `tzset()`, jak określono w strukturze `tm`. W przypadku błędu funkcja zwraca wartość `-1` (zrzutowaną na typ `time_t`).

Funkcja `ctime()` przekształca czas reprezentowany przez typ `time_t` na łańcuch ASCII:

```
#include <time.h>

char * ctime (const time_t *timep);
char * ctime_r (const time_t *timep, char *buf);
```

W przypadku błędu funkcja zwraca `NULL`. Na przykład:

```
time_t t = time (NULL);
printf ("Czas dla poprzedniej linii kodu: %s", ctime (&t));
```

Należy zwrócić uwagę na brak znaku nowej linii w poleceniu `printf`. Być może jest to niepożądany sposób działania, lecz funkcja `ctime()` dołącza znak nowej linii do zwracanego łańcucha.

Podobnie jak funkcja `asctime()`, również `ctime()` zwraca wskaźnik do łańcucha statycznego. Ponieważ to zachowanie nie jest bezpieczne w przypadku obsługi wątków, w programach wielowątkowych powinno używać się funkcji `ctime_r()`, która wykorzystuje bufor dostarczany w parametrze `buf`. Bufor powinien mieć długość co najmniej 26 znaków.

Funkcja `gmtime()` zapisuje czas reprezentowany przez typ `time_t` do struktury `tm`, dopasowując go do strefy czasowej UTC:

```
#include <time.h>

struct tm * gmtime (const time_t *timep);
struct tm * gmtime_r (const time_t *timep, struct tm *result);
```

W przypadku błędu funkcja zwraca `NULL`.

Funkcja w sposób statyczny przydziela pamięć dla zwracanej struktury i dlatego też nie obsługuje bezpiecznie wątków. W programach wielowątkowych powinno używać się funkcji `gmti`
`↪me_r()`, która wykorzystuje bufor wskazywany przez parametr `result`.

Funkcje `localtime()` oraz `localtime_r()` wykonują działania podobne odpowiednio do funkcji `gmtime()` oraz `gmtime_r()`, lecz dopasowują czas do strefy czasowej użytkownika:

```
#include <time.h>

struct tm * localtime (const time_t *timep);
struct tm * localtime_r (const time_t *timep, struct tm *result);
```

Podobnie jak w przypadku funkcji `mktime()` wywołanie `localtime()` również używa funkcji `tzset()` oraz inicjalizuje strefę czasową. W przypadku funkcji `localtime_r()` czynność ta nie jest określona.

Funkcja `difftime()` zwraca liczbę sekund (zrzutowaną na typ `double`), które upłynęły pomiędzy dwoma wartościami czasu `time_t`.

```
#include <time.h>

double difftime (time_t time1, time_t time0);
```

W przypadku systemów kompatybilnych ze standardem POSIX `time_t` jest typem arytmetycznym, stąd też wywołanie funkcji `difftime()` jest równoważne poniższemu działaniu, przy założeniu, że zignorowane zostaje wykrywanie przepełnienia podczas wykonywania operacji odejmowania:

```
(double) (time1 - time0)
```

W systemie Linux `time_t` jest typem całkowitoliczbowym, dlatego też nie jest wymagane jego rzutowanie na typ `double`. W celu zapewnienia przenośności należy jednak używać funkcji `difftime()`.

Dostrajanie zegara systemowego

Duże i gwałtowne zmiany czasu rzeczywistego mogą siać spustoszenie w aplikacjach wykonujących operacje zależne od czasu bezwzględnego. Rozważmy przykład narzędzia *make*, które tworzy projekty programistyczne, posługując się plikiem *Makefile*. Podczas każdego wywołania tego programu nie następuje kompilacja wszystkich plików źródłowych; gdyby tak było, wówczas — w przypadku dużych projektów — pojedyncza zmiana w pliku powodowałaby, że proces ponownego budowania trwałby godzinami. Zamiast tego narzędzie *make* porównuje czas modyfikacji pliku źródłowego (na przykład `wolf.c`) z czasem modyfikacji pliku obiektowego (`wolf.o`). Jeśli plik źródłowy (lub jakiekolwiek pliki zależne od niego, takie jak `wolf.h`) jest nowszy niż plik obiektowy, wówczas narzędzie *make* kompiluje ponownie plik kodu źródłowego i tworzy uaktualniony plik obiektowy. Jeśli jednak plik źródłowy nie jest nowszy od pliku obiektowego, wówczas nie zostaje podjęta żadna akcja.

Mając w pamięci powyżej przedstawiony sposób działania, należy rozważyć obecnie, co mogłoby się stać, gdyby użytkownik zauważył kilkugodzinną niezgodność czasu systemowego z czasem aktualnym i uruchomił narzędzie *date*, aby go uaktualnić, a następnie zmodyfikował i zapisał plik `wolf.c`. Gdyby użytkownik cofnął aktualny czas systemowy, wówczas mogłyby pojawić się kłopoty: plik `wolf.c` wyglądałby na starszy od pliku `wolf.o` — nawet gdyby tak w rzeczywistości nie było! Nie zostałaby wówczas wykonana operacja ponownej kompilacji pliku źródłowego.

Aby zapobiec powstawaniu tego problemu, Unix udostępnia funkcję `adjtime()`, która pozwala na powolne dostrajanie czasu aktualnego przy jednocześnie podanym kierunku zmian. Celem użycia tej funkcji jest minimalizacja zmian w systemie, które mogą zostać spowodowane przez pewne procesy działające w tle, takie jak demony NTP (Network Time Protocol), wykonujące ciągłe dostrajanie czasu poprzez korekcję przesunięcia czasowego impulsów zegarowych:

```
#define _BSD_SOURCE
#include <sys/time.h>

int adjtime (const struct timeval *delta, struct timeval *olddelta);
```

Poprawne wywołanie funkcji `adjtime()` przekazuje informację do jądra, aby spowolniło dostrajanie czasu zgodnie z podanym parametrem `delta`, a następnie zwraca `0`. Jeśli czas podany w parametrze `delta` będzie dodatni, jądro przyspieszy działanie zegara systemowego o tę wartość, dopóki poprawka nie zostanie w pełni zrealizowana. Jeśli czas podany w parametrze `delta` będzie ujemny, jądro spowolni działanie zegara systemowego o tę wartość, dopóki poprawka nie zostanie w pełni zrealizowana. Jądro realizuje wszystkie dopasowania w ten sposób, że prędkość działania zegara zwiększa się monotonicznie i dzięki temu nigdy nie pojawia się gwałtowna zmiana czasu. Nawet w przypadku ujemnej wartości parametru `delta` operacja dostrajania nie cofnie zegara systemowego; zamiast tego działanie zegara ulegnie spowolnieniu aż do momentu, gdy czas systemowy stanie się poprawny.

Jeśli parametr `delta` jest różny od `NULL`, wówczas jądro zatrzymuje realizację poprzednio zarejestrowanych poprawek. Jednak będzie zachowana ta część zmian, która już została wykonana. Jeśli wartość `olddelta` jest różna od `NULL`, wówczas każda poprzednio zarejestrowana i jeszcze niezrealizowana poprawka zostanie zapisana do dostarczonej struktury `timeval`. Przekazanie `NULL` w parametrze `delta` oraz poprawnej wartości w parametrze `olddelta` pozwala na odświeżenie trwającej właśnie korekcji.

Poprawki realizowane poprzez użycie funkcji `adjtime()` powinny być niewielkie — idealnym ich zastosowaniem są omówione wcześniej aplikacje NTP, które używają niedużych korekt (rzędu kilku sekund). Progi graniczne minimalnych i maksymalnych wartości poprawek dla dowolnego kierunku zmian są równe w przypadku systemu Linux paru tysiącom sekund.

W przypadku błędu funkcja `adjtime()` zwraca `−1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EFAULT`
: Parametry `delta` lub `olddelta` są niepoprawnymi wskaźnikami.

`EINVAL`
: Wartość dostrojenia, określona w parametrze `delta`, jest zbyt duża lub mała.

`EPERM`
: Użytkownik, wywołujący funkcję `adjtime()`, nie posiada uprawnienia `CAP_SYS_TIME`.

W protokole RFC 1305 zdefiniowano własny algorytm dostrajania zegara systemowego, który posiada znacznie więcej możliwości i jest dużo bardziej złożony niż procedura korekcji stopniowej, używana w funkcji `adjtime()`. Algorytm ten zaimplementowano w systemie Linux za pomocą funkcji systemowej `adjtimex()`:

```
#include <sys/timex.h>

int adjtimex (struct timex *adj);
```

Wywołanie funkcji `adjtimex()` powoduje odczytanie parametrów jądra związanych z czasem. Parametry zostają zapisane do struktury `timex` wskazywanej przez wskaźnik `adj`. W zależności od wartości pola `modes` znajdującego się w tej strukturze, dana funkcja systemowa może dodatkowo ustawiać pewne parametry.

Struktura `timex` zdefiniowana jest w pliku nagłówkowym `<sys/timex.h>`:

```
struct timex
{
   int modes;            /* przełącznik trybu */
   long offset;          /* przesunięcie czasu (mikrosekundy) */
   long freq;            /* przesunięcie częstotliwości (skalowane w ppm) */
   long maxerror;        /* błąd maksymalny (mikrosekundy) */
   long esterror;        /* błąd oszacowany (mikrosekundy) */
   int status;           /* status zegara */
   long constant;        /* stała czasowa PLL */
   long precision;       /* dokładność zegara (mikrosekundy) */
   long tolerance;       /* tolerancja częstotliwości zegara (ppm) */
   struct timeval time;  /* aktualny czas */
   long tick;            /* odstęp czasowy między taktami zegara (mikrosekundy) */
};
```

Pole `modes` jest sumą bitową zera lub większej liczby poniższych znaczników:

`ADJ_OFFSET`
: Ustawia przesunięcie czasu poprzez użycie pola `offset`.

`ADJ_FREQUENCY`
: Ustawia przesunięcie częstotliwości poprzez użycie pola `freq`.

`ADJ_MAXERROR`
: Ustawia maksymalną wartość błędu poprzez użycie pola `maxerror`.

`ADJ_ESTERROR`
: Ustawia błąd oszacowany poprzez użycie pola `esterror`.

`ADJ_STATUS`
: Ustawia status zegara poprzez użycie pola `status`.

`ADJ_TIMECONST`
: Ustawia stałą czasową dla pętli synchronizacji fazowej (PLL) poprzez użycie pola `constant`.

`ADJ_TICK`
: Ustawia wartość taktu zegarowego poprzez użycie pola `tick`.

`ADJ_OFFSET_SINGLESHOT`
: Ustawia jednorazowo przesunięcie częstotliwości poprzez użycie pola `freq`, wykorzystując do tego celu prosty algorytm zastosowany w funkcji `adjtime()`.

Jeśli pole `modes` jest równe zeru, nie następuje ustawienie parametrów. Tylko użytkownik, posiadający uprawnienie `CAP_SYS_TIME`, może dostarczyć do funkcji niezerową wartość pola `modes`; dowolny użytkownik może podać zero w polu `modes`, odczytując dzięki temu wartości wszystkich parametrów, lecz nie może żadnego z nich ustawić.

W przypadku sukcesu funkcja `adjtimex()` zwraca aktualny stan zegara, który jest równy jednej z poniższych wartości:

`TIME_OK`
: Zegar jest zsynchronizowany.

`TIME_INS`
: Sekunda przestępna zostanie dodana.

`TIME_DEL`
: Sekunda przestępna zostanie usunięta.

`TIME_OOP`
: Właśnie trwa sekunda przestępna.

`TIME_WAIT`
: Właśnie wystąpiła sekunda przestępna.

`TIME_BAD`
: Zegar nie jest zsynchronizowany.

W przypadku błędu funkcja `adjtimex()` zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EFAULT`
: Parametr `adj` jest niepoprawnym wskaźnikiem.

`EINVAL`
: Jedno lub więcej pól `modes`, `offset` lub `tick` nie jest poprawne.

`EPERM`
: Pole `modes` ma wartość niezerową, lecz wywołujący użytkownik nie posiada uprawnienia `CAP_SYS_TIME`.

Funkcja `adjtimex()` jest specyficzna dla Linuksa. W aplikacjach, dla których wymagana jest przenośność, należy raczej używać funkcji `adjtime()`.

W protokole RFC 1305 zdefiniowano skomplikowany algorytm, dlatego też pełna analiza funkcji `adjtimex()` nie została uwzględniona w tej książce.

Stan uśpienia i oczekiwania

Istnieje wiele funkcji, które umożliwiają procesowi przejście w stan uśpienia (wstrzymanego wykonywania) i trwanie w nim przez określony czas. Pierwsza z tych funkcji, zwana `sleep()`, pozwala na uśpienie wywołującego procesu przez zadaną liczbę sekund, przekazaną w parametrze `seconds`:

```
#include <unistd.h>

unsigned int sleep (unsigned int seconds);
```

Funkcja zwraca liczbę sekund, które *nie* zostały wykorzystane podczas stanu uśpienia. Dlatego też w przypadku poprawnego wykonania funkcja zwraca zero, lecz może również zwrócić dowolną liczbę zawierającą się w przedziale od zera do wartości parametru `seconds` włącznie (na przykład, jeśli stan uśpienia został przerwany przez pojawienie się sygnału). W większości przypadków użytkowników funkcji `sleep()` nie interesuje, jak długo dany proces był faktycznie w stanie uśpienia, dlatego też sprawdzenie wartości powrotnej nie zostaje przeprowadzone:

```
sleep (7); /* stan uśpienia przez siedem sekund */
```

Jeśli rzeczywiście ważną sprawą jest zapewnienie uśpienia procesu przez cały czas, który został przekazany do funkcji, wówczas można kontynuować wywoływanie funkcji `sleep()` z parametrem, równym wartości powrotnej z jej poprzedniego wykonania, dopóki nie zwróci ona wartości zero:

```
unsigned int s = 5;

/* przejdź w bezwarunkowy stan uśpienia na pięć sekund */
while ((s = sleep (s)))
   ;
```

Obsługa stanu uśpienia z dokładnością do mikrosekund

Wprowadzanie procesów w stan uśpienia z dokładnością do sekundy nie jest wielkim osiągnięciem. W nowoczesnych systemach sekunda jest wiecznością, dlatego też w programach często wymagana jest wyższa dokładność. Zadanie to jest realizowane za pomocą funkcji `usleep()`:

```
/* wersja BSD */
#include <unistd.h>

void usleep (unsigned long usec);

/* wersja SUSv2 */
#define _XOPEN_SOURCE 500
#include <unistd.h>

int usleep (useconds_t usec);
```

Poprawne wywołanie funkcji `usleep()` pozwala wprowadzić proces wywołujący w stan uśpienia na określony czas, którego wartość przekazana jest w parametrze `usec` i wyrażona w mikrosekundach. Niestety prototypy funkcji, zdefiniowane w systemie BSD oraz Single UNIX Specification, nie są ze sobą zgodne. Wariant BSD używa typu `unsigned long` i nie zwraca żadnej wartości. Specyfikacja SUS definiuje funkcję `usleep()` w ten sposób, iż przyjmuje ona parametr o typie `useconds_t` i zwraca liczbę `int`. W systemie Linux użyty zostanie wariant SUS, jeśli stałej `_XOPEN_SOURCE` przyporządkowana będzie liczba o wartości `500` albo większej. Jeśli stała `_XOPEN_SOURCE` nie jest zdefiniowana lub posiada wartość mniejszą od `500`, wówczas w Linuksie używana jest wersja BSD funkcji `usleep()`.

Wersja SUS zwraca `0` w przypadku sukcesu, natomiast `-1` w przypadku błędu. Poprawnymi wartościami kodów błędów są `EINTR` i `EINVAL`. Pierwsza wartość wskazuje, że stan uśpienia został przerwany przez pojawienie się sygnału, druga, iż wartość parametru `usec` jest zbyt duża (w przypadku systemu Linux poprawny jest pełen zakres dla użytego typu danych, zatem ten błąd nigdy się nie pojawi).

Zgodnie ze specyfikacją typ `useconds_t` jest typem całkowitoliczbowym bez znaku, pozwalającym na przechowanie liczb o maksymalnej wartości równej 1 000 000.

Z powodu istnienia różnic pomiędzy sprzecznymi prototypami funkcji oraz faktu, iż niektóre systemy uniksowe mogą wspierać tylko jedną z nich, rozsądnym zachowaniem jest wstrzymanie się przed jawnym użyciem typu `useconds_t` w tworzonym kodzie. W celu uzyskania maksymalnej przenośności należy założyć, że parametr posiada typ `unsigned int` oraz nie używać powrotnej wartości funkcji `usleep()`:

```
void usleep (unsigned int usec);
```

Użycie funkcji jest następujące:

```
unsigned int usecs = 200;

usleep (usecs);
```

Metoda ta jest poprawna również dla drugiego wariantu funkcji; wciąż istnieje także możliwość sprawdzenia błędów:

```
errno = 0;
usleep (1000);
if (errno)
   perror ("usleep");
```

W większości programów jednakże nie sprawdza się ani też nie dba o błędy, zwracane przez funkcję `usleep()`.

Obsługa stanu uśpienia z dokładnością do nanosekund

Użycie funkcji `usleep()` nie jest zalecane w Linuksie, ponieważ została ona zastąpiona przez funkcję `nanosleep()`, która udostępnia lepszy interfejs oraz dokładność na poziomie nanosekund:

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int nanosleep (const struct timespec *req, struct timespec *rem);
```

Poprawne wywołanie funkcji `nanosleep()` pozwala na wprowadzenie procesu wywołującego w stan uśpienia na określony czas, którego wartość przekazana jest w parametrze `req`. Następnie funkcja zwraca wartość `0`. W przypadku błędu funkcja zwraca `−1` oraz odpowiednio ustawia zmienną `errno`. Jeśli pojawi się sygnał, który przerwie stan uśpienia, wówczas działanie funkcji zakończy się przed upływem zadanego czasu. W tym przypadku funkcja `nanosleep()` zwróci `−1` oraz ustawi zmienną `errno` na wartość `EINTR`. Jeśli wartość parametru `rem` jest różna od `NULL`, wówczas funkcja umieszcza w nim informację o czasie niewykorzystanym (ilości czasu, którego proces nie zużył podczas stanu uśpienia). Program może ponownie wywołać funkcję, przekazując wartość `rem` do parametru `req` (jak zostanie to pokazane w dalszej części tego podrozdziału).

Poniżej podano możliwe wartości zmiennej `errno`:

`EFAULT`
: Parametry `req` lub `rem` są niepoprawnymi wskaźnikami.

`EINVAL`
: Jedno z pól struktury `req` jest niepoprawne.

Standardowy sposób użycia funkcji jest prosty:

```
struct timespec req =
   { .tv_sec = 0,
     .tv_nsec = 200 };

/* uśpienie przez 200 ns */
ret = nanosleep (&req, NULL);
if (ret)
   perror ("nanosleep");
```

Poniżej podano przykład użycia drugiego parametru, pozwalający na kontynuowanie stanu uśpienia w momencie jego przerwania:

```
struct timespec req =
   { .tv_sec = 0,
     .tv_nsec = 1369 };
struct timespec rem;
int ret;

/* uśpienie przez 1369 ns */
retry:
ret = nanosleep (&req, &rem);
if (ret)
{
   if (errno == EINTR)
   {
```

```
        /* ponowne wywołanie funkcji z parametrem równym pozostałemu czasowi */
        req.tv_sec = rem.tv_sec;
        req.tv_nsec = rem.tv_nsec;
        goto retry;
    }
    perror ("nanosleep");
}
```

Poniżej przedstawiono alternatywne rozwiązanie (być może bardziej efektywne, lecz mniej czytelne) tego samego problemu:

```
struct timespec req =
   { .tv_sec = 1,
     .tv_nsec = 0 };
struct timespec rem, *a = &req, *b = &rem;

/* uśpienie przez jedną sekundę */
while (nanosleep (a, b) && errno == EINTR)
{
   struct timespec *tmp = a;
   a = b;
   b = tmp;
}
```

Funkcja `nanosleep()` posiada kilka zalet w porównaniu z funkcjami `sleep()` oraz `usleep()`:

- Realizuje dokładność na poziomie nanosekund, w porównaniu z dokładnością do sekund lub mikrosekund.
- Jest zdefiniowana w standardzie POSIX.1b.
- Nie została zaimplementowana przy użyciu sygnałów (co mogłoby spowodować pewne zagrożenia, omówione w dalszej części tego rozdziału).

Pomimo wycofania oficjalnego poparcia w wielu programach wciąż używa się funkcji `usleep()` zamiast `nanosleep()` — na szczęście coraz mniej aplikacji stosuje obecnie funkcji `sleep()`.

Zaawansowane zarządzanie stanem uśpienia

Podobnie jak miało to miejsce w przypadku wszystkich typów funkcji, które dotychczas poddane zostały analizie i były związane z czasem, rodzina funkcji zegarowych, zdefiniowana w standardzie POSIX, udostępnia najbardziej zaawansowany interfejs, służący do obsługi stanu uśpienia:

```
#include <time.h>
int clock_nanosleep (clockid_t clock_id, int flags, const struct timespec *req,
  struct timespec *rem);
```

Funkcja `clock_nanosleep()` działa podobnie jak funkcja `nanosleep()`.

Weźmy pod uwagę następujące wywołanie funkcji:

```
ret = nanosleep (&req, &rem);
```

Jest równoważne poniższemu wywołaniu:

```
ret = clock_nanosleep (CLOCK_REALTIME, 0, &req, &rem);
```

Różnica polega na użyciu dodatkowych parametrów `clock_id` oraz `flags`. Pierwszy z nich określa źródło czasu, które będzie służyło do wykonania pomiaru. Dostępnych jest więcej źródeł

czasu, chociaż nie można użyć zegara CPU dla procesu wywołującego (np. CLOCK_PROCESS_
↪CPUTIME_ID); takie użycie nie miałoby sensu, ponieważ funkcja zatrzymuje wykonywanie procesu, stąd też jego czas przestałby przyrastać.

Typ źródła czasu zależny jest od rodzaju uśpienia dla danego programu. Jeśli program musi przebywać w stanie uśpienia do pewnej określonej wartości czasu bezwzględnego, wówczas najbardziej sensowne jest użycie CLOCK_REALTIME. Gdy program musi być uśpiony przez pewien okres, wówczas idealnym źródłem czasu dla niego jest na pewno CLOCK_MONOTONIC.

Parametr flags może być równy wartościom TIMER_ABSTIME lub 0. Jeśli jest równy TIMER_
↪ABSTIME, wówczas wartość określona przez parametr req traktowana jest jako absolutna, a nie względna. Rozwiązuje to ewentualny problem współzawodnictwa. Aby wyjaśnić wartość tego parametru, załóżmy, że proces w chwili czasu T_0 chce przejść do stanu uśpienia, w którym będzie trwał do chwili czasu T_1. W momencie T_0 proces wywołuje funkcję clock_gettime(), aby otrzymać wartość aktualnego czasu (T_0). Następnie odejmuje T_0 od T_1, uzyskując wartość Y, którą w dalszej kolejności przekazuje do funkcji clock_nanosleep(). Minęło jednak trochę czasu pomiędzy chwilą, w której proces otrzymał wartość aktualnego czasu a chwilą, w której przeszedł w stan uśpienia. Jeszcze większym problemem może być sytuacja, gdy proces został przeszeregowany przez zarządcę procesów, wystąpił błąd stronicowania lub podobne zdarzenie. Zawsze istnieje ewentualna możliwość powstania sytuacji współzawodnictwa pomiędzy otrzymaniem wartości aktualnego czasu, obliczeniem różnicy czasowej oraz rzeczywistym przejściem w stan uśpienia.

Użycie znacznika TIMER_ABSTIME rozwiązuje problem współzawodnictwa poprzez zezwolenie na bezpośrednie użycie przez proces wartości T_1. Jądro zatrzymuje działanie procesu, dopóki określone źródło czasu nie osiągnie wartości T_1. Jeśli aktualny czas dla danego źródła czasu przekracza wartość T_1, wówczas wywołanie funkcji kończy się natychmiast.

Poniżej przedstawione zostaną przykłady kodów używające bezwzględnych i względnych wartości czasu uśpienia. Następujący przykład powoduje wprowadzenie procesu w stan uśpienia na czas 1,5 sekundy:

```
struct timespec ts = { .tv_sec = 1, .tv_nsec = 500000000 };
int ret;

ret = clock_nanosleep (CLOCK_MONOTONIC, 0, &ts, NULL);
if (ret)
   perror ("clock_nanosleep");
```

I odwrotnie, poniższy przykład powoduje wprowadzenie procesu w stan uśpienia do momentu osiągnięcia pewnej bezwzględnej wartości czasu — w tym przypadku jednej sekundy od momentu, gdy funkcja clock_gettime(), użyta z parametrem źródła czasu CLOCK_MONOTONIC, zakończy swoje działanie:

```
struct timespec ts;
int ret;

/* przejście w stan uśpienia do momentu osiągnięcia czasu równego jednej sekundzie od TERAZ */
ret = clock_gettime (CLOCK_MONOTONIC, &ts);
if (ret)
{
   perror ("clock_gettime");
   return;
}

ts.tv_sec += 1;
```

```
printf ("Stan uśpienia ma trwać do momentu: sec=%ld nsec=%ld\n", ts.tv_sec,
↪ts.tv_nsec);
ret = clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME, &ts, NULL);
if (ret)
   perror ("clock_nanosleep");
```

Dla większości programów wymagane jest tylko ustalanie względnego czasu trwania w stanie uśpienia, ponieważ ich potrzeby w tym zakresie nie są ściśle określone. Niektóre procesy czasu rzeczywistego posiadają jednak bardzo dokładne wymagania czasowe; dla takich procesów niezbędne jest ustalenie bezwzględnej wartości czasu trwania w stanie uśpienia, aby zapobiec niebezpieczeństwu ewentualnego powstania szkodliwej sytuacji współzawodnictwa.

Przenośny sposób wprowadzania w stan uśpienia

W rozdziale 2. przedstawiono działanie przydatnej funkcji `select()`:

```
#include <sys/select.h>

int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
  struct timeval *timeout);
```

Jak wówczas wspomniano, funkcja `select()` dostarcza przenośnego sposobu na uzyskanie stanu uśpienia z dokładnością większą od sekundowej. Przez dość długi czas w przenośnych programach uniksowych używano jedynie funkcji `sleep()`, aby realizować przejście w stan uśpienia: funkcja `usleep()` nie była szeroko dostępna, natomiast `nanosleep()` nie została jeszcze stworzona. Projektanci odkryli jednak, że przekazywanie do funkcji `select()` parametru `n` równego `0`, wartości `NULL` we wszystkich trzech wskaźnikach `fd_set` oraz żądanego czasu uśpienia w parametrze `timeout` było przenośnym i efektywnym sposobem na wprowadzenie procesu w stan uśpienia:

```
struct timeval tv =
   { .tv_sec = 0,
     .tv_usec = 757 };

/* przejdź w stan uśpienia na 757 mikrosekund */
select (0, NULL, NULL, NULL, &tv);
```

Jeśli przenośność do starszych systemów uniksowych odgrywa dużą rolę, najlepszym rozwiązaniem jest użycie funkcji `select()`.

Przepełnienia

Użycie każdego z interfejsów, poddanych analizie w tym podrozdziale, gwarantuje, że stan uśpienia będzie trwać *co najmniej przez taki czas, jaki zażądano podczas jego wywołania* (w przeciwnym razie interfejsy zwrócą kod błędu). Nigdy nie zakończą się sukcesem, jeśli nie upłynie żądany czas. Jest jednakże możliwe, że żądany okres będzie trwał *dłużej*, niż wymagano.

To zjawisko może być spowodowane zwykłym działaniem szeregowania procesów — być może żądany czas już upłynął, a jądro we właściwym momencie obudziło dany proces, lecz zarządca procesów wybrał inne zadanie, które zostało następnie przez niego uruchomione.

Istnieje jednak bardziej podstępny przypadek: *przepełnienie licznika*. Problem ten pojawia się wówczas, gdy rozdzielczość licznika jest gorsza od wymaganego przedziału czasowego. Załóżmy, że licznik systemowy generuje takty zegarowe co 10 milisekund, natomiast proces wymaga uśpienia jedynie przez 1 milisekundę. System może mierzyć czas oraz odpowiadać na zda-

rzenia z nim związane (takie jak obudzenie procesu ze stanu uśpienia) jedynie co 10 milisekund. Jeśli w momencie, gdy proces wysyła żądanie uśpienia, licznik znajduje się 1 milisekundę przed wygenerowaniem taktu zegarowego, wówczas wszystko odbędzie się poprawnie — w ciągu jednej milisekundy zakończy się uśpienie procesu i jądro będzie mogło go obudzić. Jeśli jednak licznik generuje takt zegara w momencie, gdy proces żąda przejścia w stan uśpienia, wówczas w ciągu następnych 10 milisekund nie pojawi się żaden nowy takt zegarowy. Dlatego też proces będzie przebywał w stanie uśpienia przez dodatkowe 9 milisekund! Oznacza to, że przepełnienie wyniesie 9 milisekund. Licznik o okresie równym *X* posiada przeciętny współczynnik przepełnienia równy *X*/2.

Dzięki użyciu źródeł czasu o wysokiej dokładności, udostępnianych na przykład przez liczniki POSIX, a także stosowaniu wyższych wartości `HZ`, można zminimalizować powstawanie zjawiska przepełnienia licznika.

Alternatywy stanu uśpienia

Jeśli to tylko możliwe, należy unikać przechodzenia w stan uśpienia. Często nie da się jednak tego uniknąć, lecz nie jest to kłopotliwe, szczególnie, gdy kod nie działa przez czas krótszy od sekundy. Program, który często przechodzi w stan uśpienia, aby w „aktywny sposób" oczekiwać na zdarzenia[4], jest źle zaprojektowany. Lepszy kod posiada taki program, który blokuje się przy obsłudze deskryptora pliku, pozwalając jądru na wprowadzenie procesu w stan uśpienia, a następnie jego obudzenie. Zamiast uruchamiać proces w pętli, dopóki nie pojawi się oczekiwane zdarzenie, jądro może zatrzymać jego wykonanie i obudzić go dopiero wtedy, gdy będzie to wymagane.

Liczniki

Liczniki udostępniają mechanizm pozwalający na wysłanie powiadomienia do procesu, gdy upłynie określony czas. Zanim licznik zakończy swoje działanie, mija czas zwany *opóźnieniem* lub *czasem wygaśnięcia* licznika. Od rodzaju licznika zależy, w jaki sposób po jego wygaśnięciu następuje powiadomienie procesu przez jądro. Jądro Linuksa oferuje kilka typów liczników. Zostaną one poddane analizie w kolejnych podrozdziałach.

Liczniki są przydatne z kilku powodów. Przykładami ich użycia jest odświeżanie ekranu 60 razy na sekundę lub przerwanie oczekującej transakcji, jeśli wciąż nie zakończyła się ona po upływie 500 milisekund.

Proste alarmy

Funkcja `alarm()` jest najprostszym interfejsem licznikowym:

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
```

[4] Oczekiwanie to polega na przepytywaniu zdarzeń w pętli, która zawiera również polecenie krótkotrwałego przejścia w stan uśpienia. Pozwala to zarządcy procesów na obsługę przeszeregowania podczas trwania tej pętli — *przyp. tłum.*

Wywołanie powyższej funkcji umożliwia dostarczenie sygnału SIGALRM do wywołującego procesu po upływie czasu rzeczywistego, przekazanego w parametrze seconds i wyrażonego w sekundach. Jeśli wcześniej zaszeregowany sygnał jest w trakcie oczekiwania, funkcja przerywa działanie tego alarmu i zastępuje go nowo zdefiniowanym, zwracając liczbę sekund, które pozostały do uruchomienia poprzedniego. Jeśli parametr seconds wynosi zero, wówczas ewentualny wcześniejszy alarm zostanie usunięty, a nowy alarm nie będzie zaszeregowany.

Dlatego też poprawne użycie tej funkcji wymaga zarejestrowania procedury obsługi dla sygnału SIGALRM (sygnały i procedury ich obsługi zostały omówione w poprzednim rozdziale).

Poniżej przedstawiono fragment kodu, który rejestruje procedurę obsługi sygnału SIGALRM o nazwie alarm_handler() oraz definiuje alarm o opóźnieniu równym pięciu sekundom:

```
void alarm_handler (int signum)
{
   printf ("Minęło pięć sekund!\n");
}

void func (void)
{
   signal (SIGALRM, alarm_handler);
   alarm (5);

   pause ( );
}
```

Liczniki interwałowe

Funkcje systemowe obsługujące *liczniki interwałowe* (ang. *interval timers*) pojawiły się po raz pierwszy w systemie 4.2BSD, a potem zostały zdefiniowane w standardzie POSIX. Pozwalają na uzyskanie większych możliwości kontroli niż użycie funkcji alarm():

```
#include <sys/time.h>

int getitimer (int which, struct itimerval *value);

int setitimer (int which, const struct itimerval *value, struct itimerval *ovalue);
```

Liczniki interwałowe zachowują się podobnie jak funkcja alarm(), lecz opcjonalnie mogą zostać w sposób automatyczny ponownie zainicjalizowane i działają w trzech różnych trybach:

ITIMER_REAL
: Licznik mierzy upływ czasu rzeczywistego. Gdy upłynie zadany czas, wówczas jądro wysyła do procesu sygnał SIGALRM.

ITIMER_VIRTUAL
: Licznik jest zmniejszany jedynie wówczas, gdy następuje wykonanie kodu procesu z przestrzeni użytkownika. Gdy upłynie zadany czas, wówczas jądro wysyła do procesu sygnał SIGVTALRM.

ITIMER_PROF
: Licznik jest zmniejszany, gdy następuje wykonanie kodu procesu, jak i wówczas, gdy jądro przeprowadza działania, związane z danym procesem (na przykład, realizując wywołanie funkcji systemowej). Gdy upłynie zadany czas, wówczas jądro wysyła do procesu sygnał SIGVPROF. Tryb ten jest zwykle połączony z trybem ITIMER_VIRTUAL, co pozwala w programie na pomiar zużytego czasu dla procesu działającego w przestrzeni użytkownika i w przestrzeni jądra.

Użycie trybu ITIMER_REAL pozwala na pomiar tego samego czasu jak w przypadku funkcji alarm(); pozostałe dwa tryby przydatne są podczas wykonywania operacji profilowania.

Struktura itimerval pozwala użytkownikowi na określenie przedziału czasowego, po którym nastąpi wygaśnięcie licznika, jak również nowej wartości czasu służącej do ponownego zainicjalizowania licznika po jego wyzerowaniu:

```
struct itimerval
{
   struct timeval it_interval; /* wartość następna */
   struct timeval it_value; /* wartość aktualna */
};
```

Elementami struktury itimerval są wcześniej zaprezentowane struktury timeval. Pozwalają one na uzyskanie dokładności na poziomie mikrosekund:

```
struct timeval
{
   long tv_sec; /* sekundy */
   long tv_usec; /* mikrosekundy */
};
```

Funkcja setitimer() definiuje licznik o typie określonym w parametrze which, ustalając czas wygaśnięcia w polu it_value. Gdy upłynie czas określony w polu it_value, jądro ponownie inicjalizuje licznik, przekazując jako czas wygaśnięcia wartość pola it_interval. Jeśli licznik wygaśnie, a wartość it_interval będzie równa zeru, wówczas nie zostanie on ponownie zainicjalizowany. Podobnie, jeśli dla aktywnego licznika nastąpi przekazanie zera w polu it_value, wówczas zostanie on zatrzymany i nie będzie ponownie zainicjalizowany.

Jeśli parametr ovalue jest różny od NULL, wówczas zawiera on poprzednie wartości licznika interwałowego o typie which.

Funkcja getitimer() zwraca aktualne wartości dla licznika interwałowego o typie which.

Obie funkcje zwracają zero w przypadku sukcesu, natomiast −1 w przypadku błędu, ustawiając wówczas zmienną errno na jedną z poniższych wartości:

EFAULT
: Parametry value lub ovalue są niepoprawnymi wskaźnikami.

EINVAL
: Parametr which nie jest poprawnym typem licznika interwałowego.

Poniższy fragment kodu tworzy procedurę obsługi sygnału SIGALRM (szczegóły w rozdziale 9.), a następnie inicjalizuje licznik interwałowy z początkowym czasem wygaśnięcia równym pięciu sekundom oraz kolejnym czasem wygaśnięcia równym jednej sekundzie:

```
void alarm_handler (int signo)
{
   printf ("Wygaśnięcie licznika!\n");
}

void foo (void)
{
   struct itimerval delay;
   int ret;

   signal (SIGALRM, alarm_handler);

   delay.it_value.tv_sec = 5;
   delay.it_value.tv_usec = 0;
```

```
    delay.it_interval.tv_sec = 1;
    delay.it_interval.tv_usec = 0;
    ret = setitimer (ITIMER_REAL, &delay, NULL);
    if (ret)
    {
       perror ("setitimer");
       return;
    }

    pause ( );
}
```

W niektórych systemach uniksowych funkcje `sleep()` oraz `usleep()` wykorzystują sygnał `SIGALRM`, a funkcje `alarm()` i `setitimer()` używają `SIGALARM`. Dlatego też programiści muszą uważać, aby nie spowodować nakładania się tych funkcji, gdyż w przeciwnym razie rezultaty ich działań mogą być nieokreślone. Aby zrealizować krótkotrwałe oczekiwanie, programiści powinni stosować funkcję `nanosleep()`, która zgodnie ze standardem POSIX nie używa sygnałów. W przypadku liczników należy używać funkcji `settimer()` lub `alarm()`.

Liczniki zaawansowane

Interfejsy licznikowe o największych możliwościach pochodzą — co nie jest zaskoczeniem — z rodziny zegarów POSIX.

W licznikach opartych na zegarach POSIX czynnościom tworzenia, inicjalizowania oraz usuwania licznika odpowiadają trzy różne funkcje: `timer_create()` tworzy licznik, `timer_settime()` inicjalizuje licznik, a `timer_delete()` pozwala na jego usunięcie.

Rodzina zegarów POSIX związana z interfejsami licznikowymi jest niewątpliwie nie tylko najbardziej zaawansowana, lecz również najnowsza (a więc najmniej przenośna) i najbardziej skomplikowana w użyciu. Jeśli najważniejszym celem w programie będzie zachowanie przenośności lub prostoty działania, wówczas powinna zostać wybrana funkcja `settimer()`.

Tworzenie licznika

Aby stworzyć licznik, należy użyć funkcji `timer_create()`:

```
#include <signal.h>
#include <time.h>

int timer_create (clockid_t clockid, struct sigevent *evp, timer_t *timerid);
```

Poprawne wywołanie funkcji `timer_create()` tworzy nowy licznik związany z zegarem POSIX, przekazanym w parametrze `clockid`, zapisuje unikalny identyfikator licznika w parametrze `timerid` oraz zwraca `0`. Funkcja ta zaledwie przygotowuje warunki pozwalające na uruchomienie licznika; w rzeczywistości nic się nie stanie, dopóki licznik nie zostanie zainicjalizowany, co zaprezentowano w następnym podrozdziale.

Poniższy przykład tworzy nowy licznik sterowany zegarem POSIX o typie `CLOCK_PROCESS_CPUTIME_ID`, a następnie zapamiętuje jego identyfikator w zmiennej `timer`:

```
timer_t timer;
int ret;

ret = timer_create (CLOCK_PROCESS_CPUTIME_ID, NULL, &timer);
```

```
if (ret)
   perror ("timer_create");
```

W przypadku błędu funkcja zwraca `-1`, parametr `timerid` pozostaje niezdefiniowany, a zmienna `errno` zostaje ustawiona na jedną z poniższych wartości:

`EAGAIN`
: System nie posiada wystarczających zasobów, aby zakończyć tę operację.

`EINVAL`
: Zegar POSIX, określony w parametrze `clockid`, jest nieprawidłowy.

`ENOTSUP`
: Zegar POSIX, określony w parametrze `clockid`, jest prawidłowy, lecz system nie pozwala na jego użycie dla liczników. W standardzie POSIX zapewniono, że zegar `CLOCK_REALTIME` powinien być zawsze wspierany przez wszystkie implementacje służące do obsługi liczników. Od konkretnej implementacji zależy jednak, czy inne typy zegarów będą również posiadały wsparcie.

Jeśli parametr `evp` jest różny od `NULL`, wówczas definiuje asynchroniczne powiadomienie, które zostanie wygenerowane, gdy nastąpi wygaśnięcie licznika. Struktura ta jest zdefiniowana w pliku nagłówkowym `<signal.h>`. Założono, że jej zawartość powinna być nieprzejrzysta dla programisty, lecz mimo to składa się co najmniej z poniższych pól:

```
#include <signal.h>

struct sigevent
{
   union sigval sigev_value;
   int sigev_signo;
   int sigev_notify;
   void (*sigev_notify_function)(union sigval);
   pthread_attr_t *sigev_notify_attributes;
};

union sigval
{
   int sival_int;
   void *sival_ptr;
};
```

Dzięki licznikom opartym na zegarach POSIX można uzyskać dużo większe możliwości kontroli dotyczące sposobu, w jaki jądro powiadamia proces o ich wygaśnięciu. Pozwalają one procesowi na określenie, jaki dokładnie sygnał powinien zostać wygenerowany przez jądro lub nawet umożliwiają jądru stworzenie nowego wątku oraz uruchomienie funkcji w odpowiedzi na zatrzymanie się licznika. Proces określa sposób działania w momencie wygaśnięcia licznika poprzez użycie parametru `sigev_notify`, który może być równy jednej z poniżej przedstawionych wartości:

`SIGEV_NONE`
: Powiadomienie „zerowe". Podczas wygaśnięcia licznika nic się nie dzieje.

`SUGEV_SIGNAL`
: Podczas wygaśnięcia licznika jądro wysyła procesowi sygnał określony w polu `sigev_`
`↪signo`. W procedurze obsługi sygnału pole `si_value` zostaje ustawione na wartość `sigev_`
`↪value`.

SIGEV_THREAD
: W przypadku wygaśnięcia licznika jądro uruchamia nowy wątek (wewnątrz aktualnego procesu), który musi wywołać funkcję `sigev_notify_function`, przekazując wartość `sigev_value` w jej jedynym parametrze. Wątek kończy swoje działanie, gdy powraca z wywołania tej funkcji. Jeśli pole `sigev_notify_attibutes` jest różne od `NULL`, wówczas dostarczona struktura `pthread_atrtr_t` definiuje sposób działania nowego wątku.

Jeśli parametr `evp` jest równy `NULL` (jak zostało to użyte w poprzednim przykładzie), wówczas powiadomienie o wygaśnięciu licznika zostanie skonfigurowane tak, jak gdyby polu `sigev_notify` przyporządkowana została wartość `SIGEV_SIGNAL`, pole `sigev_signo` było równe `SIGALRM`, a pole `sigev_value` było identyfikatorem licznika. Dlatego też domyślnie liczniki te generują powiadomienie w sposób podobny do liczników interwałowych POSIX. Dzięki istniejącym możliwościom dopasowania do własnych potrzeb pozwalają one programistom na zrealizowanie dużo bardziej zaawansowanych zadań.

Poniższy przykład tworzy licznik używający zegara `CLOCK_REALTIME`. Gdy licznik wygaśnie, jądro wygeneruje sygnał `SIGUSR1` oraz ustawi wartość `si_value` na adres, w którym przechowywany jest identyfikator licznika:

```
struct sigevent evp;
timer_t timer;
int ret;

evp.sigev_value.sival_ptr = &timer;
evp.sigev_notify = SIGEV_SIGNAL;
evp.sigev_signo = SIGUSR1;
ret = timer_create (CLOCK_REALTIME, &evp, &timer);
if (ret)
   perror ("timer_create");
```

Inicjalizowanie licznika

Licznik, stworzony za pomocą funkcji `timer_create()`, jest niezainicjalizowany. Aby przypisać mu parametr czasu, po którym nastąpi jego wygaśnięcie oraz uruchomić go, należy użyć funkcji `timer_settime()`:

```
#include <time.h>

int timer_settime (timer_t timerid, int flags, const struct itimerspec *value,
  struct itimerspec *ovalue);
```

Poprawne wywołanie funkcji `timer_settime()` inicjalizuje licznik, określony w parametrze `timerid`, przypisując mu czas wygaśnięcia, który zdefiniowany jest w strukturze `itimerspec`:

```
struct itimerspec
{
   struct timespec it_interval; /* wartość następna */
   struct timespec it_value; /* wartość aktualna */
};
```

Podobnie jak ma to miejsce w przypadku funkcji `setitimer()`, wartość `it_value` określa aktualny czas wygaśnięcia licznika. Gdy następuje wygaśnięcie licznika, pole `it_value` zostaje ponownie zainicjalizowane wartością `it_interval`. Jeśli pole `it_interval` będzie równe `0`, wówczas licznik przestanie być licznikiem interwałowym i zakończy swoje działanie, gdy tylko upłynie czas zdefiniowany w polu `it_value`.

Elementami struktury itimerspec są wcześniej zaprezentowane struktury timespec. Pozwalają one na uzyskanie dokładności na poziomie nanosekund:

```
struct timespec
{
   time_t tv_sec; /* sekundy */
   long tv_nsec; /* nanosekundy */
};
```

Jeśli parametr flags jest równy TIMER_ABSTIME, wówczas czas określony w parametrze value jest czasem bezwzględnym (w przeciwieństwie do interpretacji domyślnej, dla której wartość ta jest względna w stosunku do aktualnego czasu). Ten zmodyfikowany sposób działania chroni przed powstaniem problemu współzawodnictwa podczas odczytywania aktualnego czasu, obliczania względnej różnicy między nim a wymaganym czasem przyszłym, a także podczas inicjalizowania licznika. Szczegóły przedstawione są we wcześniejszym podrozdziale, zatytułowanym Zaawansowane zarządzanie stanem uśpienia.

Jeśli wartość ovalue jest różna od NULL, wówczas poprzedni czas wygaśnięcia licznika zostaje zapisany do dostarczonej struktury itimerspec. Gdy licznik przestał już poprzednio działać, wtedy wartości elementów struktury zostają ustawione na 0.

Poniższy przykład używa wartości timer, która została wcześniej zainicjalizowana przy użyciu funkcji timer_create(). Prezentuje on utworzenie licznika okresowego, którego czas wygaśnięcia wynosi jedną sekundę:

```
struct itimerspec ts;
int ret;

ts.it_interval.tv_sec = 1;
ts.it_interval.tv_nsec = 0;
ts.it_value.tv_sec = 1;
ts.it_value.tv_nsec = 0;

ret = timer_settime (timer, 0, &ts, NULL);
if (ret)
   perror ("timer_settime");
```

Odczytywanie czasu wygaśnięcia licznika

Czas wygaśnięcia licznika można odczytać w każdym momencie bez potrzeby jego ponownego inicjalizowania. Do tego celu służy funkcja timer_gettime():

```
#include <time.h>

int timer_gettime (timer_t timerid, struct itimerspec *value);
```

Poprawne wywołanie funkcji timer_gettime() zapisuje do struktury value czas wygaśnięcia licznika, określonego w parametrze timerid, a następnie zwraca 0. W przypadku błędu funkcja zwraca −1 oraz odpowiednio ustawia zmienną errno na którąś z poniższych wartości:

EFAULT
: Parametr value jest niepoprawnym wskaźnikiem.

EINVAL
: Parametr timerid jest niepoprawnym zegarem.

Na przykład:

```
struct itimerspec ts;
int ret;
```

```
ret = timer_gettime (timer, &ts);
if (ret)
   perror ("timer_gettime");
else
{
   printf ("Aktualne wartości: sekundy=%ld nanosekundy=%ld\n", ts.it_value.tv_sec,
  ts.it_value.tv_nsec);
   printf ("Kolejne wartości: sekundy=%ld nanosekundy=%ld\n", ts.it_interval.tv_sec,
  ts.it_interval.tv_nsec);
}
```

Odczytywanie wartości przepełnienia licznika

W standardzie POSIX zdefiniowano interfejs służący do określenia, ile przepełnień wystąpiło dla danego licznika (jeśli w ogóle jakieś nastąpiły):

```
#include <time.h>

int timer_getoverrun (timer_t timerid);
```

W przypadku sukcesu, funkcja `timer_getoverrun()` zwraca liczbę dodatkowych zdarzeń wygaśnięcia licznika, które wystąpiły pomiędzy początkowym wygaśnięciem licznika a wysłaniem powiadomienia do procesu (na przykład, poprzez wygenerowanie sygnału), że działanie licznika się zakończyło. Na przykład, dla wcześniej zaprezentowanego kodu, w którym zdefiniowano licznik o czasie 1 milisekundy, działający przez 10 milisekund, funkcja zwróci wartość `9`.

Jeśli liczba przepełnień będzie równa lub większa od `DELAYTIMER_MAX`, wówczas funkcja zwróci wartość tej stałej.

W przypadku niepowodzenia, funkcja zwraca `−1` oraz ustawia zmienną `errno` na wartość `EINVAL`, informującą o jedynym możliwym przypadku błędu, oznaczającym, że identyfikator licznika przekazany w parametrze `timerid` jest nieprawidłowy.

Na przykład:

```
int ret;

ret = timer_getoverrun (timer);
if (ret == -1)
   perror ("timer_getoverrun");
else if (ret == 0)
   printf ("Nie wystąpiło żadne przepełnienie\n");
else
   printf ("Liczba przepełnień: %d\n", ret);
```

Usuwanie licznika

Usuwanie licznika jest prostą czynnością:

```
#include <time.h>

int timer_delete (timer_t timerid);
```

Poprawne wywołanie funkcji `timer_delete()` usuwa licznik, którego identyfikator przekazany jest w parametrze `timerid`, a następnie zwraca `0`. W przypadku błędu, funkcja zwraca `−1` oraz ustawia zmienną `errno` na wartość `EINVAL`, informującą o jedynym możliwym przypadku błędu, oznaczającym, że identyfikator licznika przekazany w parametrze `timerid` jest nieprawidłowy.

DODATEK A

Rozszerzenia kompilatora GCC dla języka C

Narzędzie GNU Compiler Collection (GCC) dostarcza wielu rozszerzeń dla języka C. Część z nich udowodniło swoją szczególną przydatność dla programistów systemowych. Większość rozszerzeń języka C, które zostaną omówione w tym dodatku, pozwala programistom na dostarczenie dalszych informacji do kompilatora, dotyczących zachowania oraz zamierzonego sposobu użycia ich kodu. Kompilator wykorzystuje te informacje w celu utworzenia bardziej efektywnego kodu maszynowego. Inne rozszerzenia uzupełniają luki w języku C, szczególnie na jego niższych poziomach.

GCC dostarcza pewnych rozszerzeń, które obecnie dostępne są już w najnowszym standardzie języka C — ISO C99. Niektóre z tych rozszerzeń działają podobnie jak ich pokrewne funkcje, zdefiniowane w standardzie C99, lecz generalnie w ISO C99 zaimplementowano inne rozszerzenia w odmienny sposób. W nowo tworzonym kodzie należy stosować warianty tych rozszerzeń, pochodzące ze standardu ISO C99. Nie będą one omówione; analizie zostaną poddane jedynie te uzupełnienia, które są specyficzne dla kompilatora GCC.

GNU C

Odmiana języka C wspierana przez kompilator GCC zwana jest często GNU C. W latach 90. XX wieku język GNU C uzupełnił pewne luki istniejące w standardzie C, dostarczając takich możliwości, jak zmienne złożone, tablice o zerowej długości, funkcje wplatane (inline) oraz inicjalizatory nazwane. Po upływie prawie dziesięciu lat język C został odnowiony i rozszerzenia GNU C stały się mniej istotne w porównaniu ze standardem ISO C99. Mimo tego język GNU C w dalszym ciągu dostarcza przydatnych opcji, a wielu programistów Linuksa wciąż używa podzbioru języka GNU C (często tylko jednego lub dwóch określonych rozszerzeń) w swoim kodzie, który jest zgodny ze standardami C90 lub C99.

Jednym ze znaczących przykładów kodu specyficznego dla kompilatora GCC jest jądro Linuksa, które zostało napisane wyłącznie w języku GNU C. Ostatnio firma Intel podjęła pewne wysiłki, aby udostępnić kompilatorowi Intel C Compiler (ICC) możliwość rozpoznawania rozszerzeń GNU C, używanych przez jądro. Zgodnie z tym wiele rozszerzeń przestaje być specyficznymi tylko dla kompilatora GCC.

Funkcje wplatane (inline)

Kompilator kopiuje cały kod funkcji „wplatanej" (ang. *inline function*) w miejsce, w którym zostaje ona wywołana. Zamiast przechowywania funkcji w oddzielnym obszarze i przeprowadzania operacji skoku do niego, gdy funkcja zostaje wywołana, jej kod jest bezpośrednio wykonywany. Taki sposób działania pozwala na uniknięcie kosztów wywołania funkcji oraz umożliwia uzyskanie ewentualnych optymalizacji w miejscu wykonania tego kodu, ponieważ kompilator może objąć działaniem zarówno procedurę wywoływaną, jak i wywołującą. Ten ostatni punkt jest szczególnie ważny, jeśli w miejscu wywołania funkcji jej parametry są stałe. Oczywiście kopiowanie zawartości funkcji do każdego miejsca kodu, w którym zostaje ona wywoływana, może spowodować przyrost jego rozmiaru. Dlatego też funkcje powinny być wplatane jedynie wówczas, gdy są one niewielkie i proste bądź nie są wywoływane z wielu różnych miejsc programu.

Przez wiele lat kompilator GCC udostępniał słowo kluczowe `inline`, które pozwalało na wplecenie danej funkcji. Standard C99 nadał mu formalny charakter:

```
static inline int foo (void) { /*… */ }
```

Technicznie rzecz ujmując, słowo to jest jedynie wskazówką — sugestią dla kompilatora, aby rozważył możliwość wplecenia danej funkcji. GCC dostarcza dalszego rozszerzenia, pozwalającego nakazać kompilatorowi wykonanie *bezwzględnego* wplecenia podanej funkcji:

```
static inline __attribute__ ((always_inline)) int foo (void) { /*… */ }
```

Najbardziej oczywistym kandydatem dla przekształcenia w funkcję wplataną jest makro preprocesora. Będzie ono działać równie dobrze, będąc funkcją wplataną, a dodatkowo zostanie wyposażone w kontrolę typów. Weźmy pod uwagę następujące makro:

```
#define max(a,b) ({ a > b ? a : b; })
```

Może ono zostać zamienione na poniższą, równoważną funkcję wplataną:

```
static inline max (int a, int b)
{
   if (a > b)
      return a;
   return b;
}
```

Programiści mają tendencję do nadużywania funkcji wplatanych. Koszt wywołania funkcji w większości nowoczesnych architektur — szczególnie w architekturze x86 — jest bardzo mały. Wybór funkcji wplatanej powinien być bardzo rozważny!

Zapobieganie wplataniu funkcji

W trybie najwyższego poziomu optymalizacji kompilator GCC automatycznie wyszukuje funkcje, które są odpowiednie dla wykonania operacji wplatania i realizuje ją. Zazwyczaj jest to dobre rozwiązanie, lecz czasem programiści zdają sobie sprawę z tego, że funkcja będzie działać niepoprawnie, jeśli zostanie przekształcona w funkcję wplataną. Możliwym przykładem takiego zachowania jest użycie słowa kluczowego `__builtin_return_address` (omówionego w dalszej części tego dodatku). Aby zapobiec wpleceniu funkcji, należy użyć słowa kluczowego `noinline`:

```
__attribute__ ((noinline)) int foo (void) { /*… */ }
```

Funkcje czyste (pure)

Funkcja „czysta" (ang. *pure function*) jest funkcją, której działanie nie powoduje żadnych zmian, a zwracana wartość odnosi się wyłącznie do jej parametrów lub nieulotnych zmiennych globalnych. Dostęp do parametrów lub zmiennych globalnych może być realizowany jedynie w trybie tylko do odczytu. W takich funkcjach mogą być zastosowane optymalizacje pętli oraz redukcje podwyrażeń. Funkcje są oznaczane jako czyste przy użyciu słowa kluczowego `pure`:

```
__attribute__ ((pure)) int foo (int val) { /*… */ }
```

Znanym przykładem funkcji czystej jest `strlen()`. W przypadku takich samych parametrów zwracana wartość funkcji nie zmienia się podczas jej wielu wywołań, dlatego też może zostać usunięta poza pętlę i jednokrotnie wykonana. Prześledźmy poniższy kod:

```
/* drukuj znak po znaku zawartość łańcucha 'p', zamieniając litery na duże */
for (i = 0; i < strlen (p); i++)
   printf ("%c", toupper (p[i]));
```

Gdyby kompilator nie miał informacji, że funkcja `strlen()` jest czysta, mógłby ją wywoływać podczas każdej iteracji w pętli!

Inteligentni programiści (a także kompilator — w przypadku, gdyby funkcja `strlen()` była oznaczona jako czysta) mogliby stworzyć następujący kod:

```
size_t len;

len = strlen (p);
for (i = 0; i < len; i++)
   printf ("%c", toupper (p[i]));
```

Jeszcze bardziej inteligentni programiści (tacy jak Czytelnicy tej książki) mogliby od niechcenia napisać poniższy kod:

```
while (*p)
   printf ("%c", toupper (*p++));
```

Nieprawidłowym działaniem, które zresztą nie ma sensu, jest zwracanie przez funkcję czystą wartości `void`, ponieważ zwracana wartość jest jedynym widocznym rezultatem wykonania takiej funkcji.

Funkcje stałe

Funkcja „stała" (ang. *constant function*) jest bardziej rygorystyczną wersją funkcji czystej. Funkcje takie nie mogą uzyskać dostępu do zmiennych globalnych oraz używać wskaźników jako swoich parametrów. Dlatego też wartość powrotna funkcji stałej związana jest wyłącznie z jej parametrami. Parametry te przekazywane są do funkcji wyłącznie przez wartość. Dla takich funkcji możliwe są dodatkowe optymalizacje, wśród których znajdują się również te dozwolone dla funkcji czystych. Funkcje matematyczne, takie jak `abs()`, są przykładami funkcji stałych (zakładając, że nie zapamiętują swoich stanów i nie wykonują żadnych sztuczek w ramach optymalizacji). Programista może oznaczyć funkcję stałą przy użyciu słowa kluczowego `const`:

```
__attribute__ ((const)) int foo (int val) { /*… */ }
```

Podobnie jak ma to miejsce w przypadku funkcji czystych, zwracanie przez funkcje stałe wartości `void` nie ma sensu.

Funkcje, które nie wracają do procedury wywołującej

Jeśli funkcja nie wraca do procedury wywołującej (być może z powodu wywołania funkcji `exit()`), wówczas programista może poinformować kompilator o tym fakcie, oznaczając ją za pomocą słowa kluczowego `noreturn`:

```
__attribute__ ((noreturn)) void foo (int val) { /*… */ }
```

Z kolei kompilator może wykonać dodatkowe optymalizacje, zakładając, że funkcja nigdy nie wróci do procedury, z której została wywołana. Zwracanie przez takie funkcje wartości różnych od `void` nie ma sensu.

Funkcje przydzielające pamięć

Jeśli funkcja zwraca wskaźnik, który nigdy nie może zostać aliasem[1] istniejącego obszaru pamięci (najprawdopodobniej wówczas, gdy funkcja właśnie przydzieliła nową pamięć i zwraca do niej wskaźnik), wówczas może ona zostać oznaczona słowem kluczowym `malloc`, dzięki czemu kompilator będzie mógł przeprowadzić odpowiednie optymalizacje:

```
__attribute__ ((malloc)) void * get_page (void)
{
   int page_size;

   page_size = getpagesize ( );
   if (page_size <= 0)
      return NULL;

   return malloc (page_size);
}
```

Wymuszanie sprawdzania wartości powrotnej dla procedur wywołujących

Atrybut `warn_unused_result` nie jest opcją optymalizacyjną, lecz elementem wspomagającym programowanie. Nakazuje on kompilatorowi, aby generował ostrzeżenie, gdy tylko wartość powrotna funkcji nie będzie zachowana lub użyta w instrukcji warunkowej:

```
__attribute__ ((warn_unused_result)) int foo (void) { /*… */ }
```

Pozwala to programiście na upewnienie się, że wszystkie procedury wywołujące sprawdzają i obsługują wartość powrotną z danej funkcji w przypadku, gdy jej użycie ma duże znaczenie. Funkcje posiadające znaczące, lecz często ignorowane wartości powrotne, takie jak `read()`, są bardzo dobrymi kandydatami do zastosowania tego atrybutu. Takie funkcje nie mogą zwracać wartości `void`.

[1] *Alias* obszaru pamięci powstaje w momencie, gdy dwie zmienne wskaźnikowe lub więcej wskazują na ten sam adres w pamięci. Może się to zdarzyć w prostych przypadkach, gdy wskaźnikowi przypisano wartość innego wskaźnika, a także w bardziej złożonych i mniej oczywistych okolicznościach. Jeśli funkcja zwraca adres nowo przydzielonej pamięci, nie istnieją żadne inne wskaźniki wskazujące na ten obszar.

Oznaczanie funkcji niezalecanych

Atrybut `deprecated` nakazuje kompilatorowi wygenerowanie ostrzeżenia w miejscu wywołania funkcji, gdziekolwiek zostanie ona użyta:

```
__attribute__ ((deprecated)) void foo (void) { /*… */ }
```

Pozwala to programistom na unikanie niezalecanych i przestarzałych funkcji.

Oznaczanie funkcji używanych

Czasem kompilator nie potrafi stwierdzić, że w kodzie występuje odwołanie do danej funkcji. Oznaczanie takiej funkcji za pomocą atrybutu `used` informuje kompilator, że jest ona używana przez program, pomimo tego iż nie odnaleziono do niej jawnych odwołań:

```
static __attribute__ ((used)) void foo (void) { /*… */ }
```

Dlatego też kompilator generuje wyjściowy kod asemblerowy i nie wyświetla ostrzeżenia, że dana funkcja nie została użyta. Atrybut ten jest użyteczny w przypadku, gdy funkcja statyczna zostaje wywołana wyłącznie z ręcznie napisanego kodu asemblerowego. Gdyby kompilator nie wykrył żadnego odwołania do funkcji i jednocześnie nie został poinstruowany za pomocą atrybutu `used`, wówczas wygenerowałby ostrzeżenie i mógłby poprzez zastosowanie optymalizacji ewentualnie usunąć tę funkcję.

Oznaczanie funkcji lub parametrów nieużywanych

Atrybut `unused` informuje kompilator, że dana funkcja lub parametr funkcji są nieużywane oraz nakazuje pominięcie generowania dla nich odpowiedniego komunikatu ostrzegawczego:

```
int foo (long __attribute__ ((unused)) value) { /*… */ }
```

Jest to użyteczne w przypadku, gdy kompilację przeprowadza się przy użyciu opcji `-W` lub `-Wunused` i wymagane jest przechwycenie nieużywanych parametrów lub gdy istnieją funkcje, które muszą odpowiadać pewnemu wcześniej zdefiniowanemu schematowi (jest to powszechne zjawisko podczas programowania interfejsów graficznych sterowanych zdarzeniami lub procedur obsługi sygnałów).

Pakowanie struktury

Atrybut `packed` informuje kompilator, że typ lub zmienna powinny zostać spakowane w pamięci przy użyciu minimalnej ilości dostępnej pamięci, ewentualnie lekceważąc wymagania wyrównania. Jeśli ten atrybut zostanie użyty razem ze słowami kluczowymi `struct` lub `union`, wówczas wszystkie zmienne będące elementami tego typu zostaną spakowane. Jeśli atrybut `packed` zostanie użyty wyłącznie dla jednej zmiennej, wówczas tylko ten określony obiekt będzie spakowany.

Następująca instrukcja pakuje wszystkie zmienne wewnątrz danej struktury, wykorzystując do tego celu minimalną ilość pamięci:

```
struct __attribute__ ((packed)) foo { ... };
```

W przykładowej strukturze zawierającej zmienną typu `char`, po której występowałoby pole o typie `int`, zmienna całkowitoliczbowa nie byłaby najprawdopodobniej umieszczona zaraz po zmiennej `char`, lecz np. po trzech dodatkowych bajtach uzupełniających. Kompilator wyrównuje zmienne, wstawiając pomiędzy nieużywane bajty uzupełnienia. Struktura spakowana nie posiada obszarów uzupełnień, przez co ewentualnie zużywa mniej pamięci, lecz nie odpowiada wymaganiom danej architektury, związanym z wyrównaniem.

Zwiększanie wartości wyrównania dla zmiennej

Kompilator GCC pozwala oprócz pakowania również na alternatywne określenie minimalnej wielkości wyrównania dla danej zmiennej. Kompilator przystąpi wówczas do wyrównania danej zmiennej, używając do tego celu *co najmniej* tylu bajtów, ile zostało mu przekazanych. Jest to zachowanie przeciwne do minimalnego wymogu wyrównania, narzucanego przez architekturę oraz ABI. Na przykład, poniższa instrukcja deklaruje zmienną o nazwie `beard_length`, posiadającą wyrównanie o wielkości co najmniej 32 bajtów (w przeciwieństwie do typowego wyrównania o wielkości 4 bajtów dla maszyn obsługujących 32-bitowe typy całkowitoliczbowe):

```
int beard_length __attribute__ ((aligned (32))) = 0;
```

Wymuszenie wartości wyrównania dla danego typu jest zwykle użyteczne jedynie wtedy, gdy obsługiwany jest sprzęt, który może narzucać większe wymagania związane z wyrównaniem, niż czyni to sama architektura lub gdy ręcznie łączy się kod języka C z kodem asemblerowym i należy użyć instrukcji, które wymagają specjalnych wartości wyrównania. Jednym z przykładów, w którym zostaje użyta taka funkcjonalność wyrównania, jest przechowywanie często używanych zmiennych w pamięci podręcznej procesora, aby zoptymalizować jej sposób działania. To rozwiązanie jest używane w jądrze Linuksa.

Alternatywą dla ustalenia pewnego minimalnego wyrównania jest wysłanie żądania do kompilatora GCC, aby wykonał wyrównanie typu do największej standardowej wartości, która kiedykolwiek została użyta dla dowolnego rodzaju danych. Na przykład, poniższy kod nakazuje kompilatorowi GCC, aby wyrównał zmienną `parrot_height` do największej wartości, jaką może użyć, która prawdopodobnie równa jest wyrównaniu do wielkości typu `double`:

```
short parrot_height __attribute__ ((aligned)) = 5;
```

Wykonanie tej instrukcji wiąże się przeważnie z pewnym kompromisem: zmienne, wyrównane w ten sposób, zużywają więcej pamięci, lecz ich kopiowanie (a także inne, złożone operacje na nich) może być szybsze, ponieważ kompilator będzie mógł wygenerować pewne instrukcje maszynowe, które sprawnie działają w przypadku większych obszarów pamięci.

Różne aspekty architektury lub systemowy zestaw narzędzi mogą narzucać maksymalne ograniczenia w przypadku ustalania wartości wyrównania zmiennych. Na przykład, w niektórych architekturach Linuksa linker nie potrafi właściwie rozpoznać wyrównań, których wartości znajdują się poza dość wąskim zakresem domyślnym. W tym przypadku wyrównanie, zrealizowane za pomocą wspomnianego powyżej słowa kluczowego, zostaje zaokrąglone do najmniejszej dopuszczalnej wartości wyrównania. Jeśli przykładowo zadane zostało wykonanie wyrównania o wartości 32, lecz linker systemowy nie potrafił zrealizować wyrównania większego niż 8 bajtów, wówczas zmienna została wyrównana do granicy obszarów o wielkości 8 bajtów.

Umieszczanie zmiennych globalnych w rejestrach

Kompilator GCC pozwala programistom na umieszczanie zmiennych globalnych w określonych rejestrach maszynowych, w których będą się one znajdować podczas działania programu. Takie zmienne zwane są *globalnymi zmiennymi rejestrowymi* (ang. *global register variables*).

Składnia języka wymaga podania rejestru maszynowego. W poniższym przykładzie użyto rejestru `ebx`:

```
register int *foo asm ("ebx");
```

Wybrana zmienna nie może być modyfikowana przez żadną funkcję. Oznacza to, że musi nadawać się do użycia przez funkcje lokalne, a oprócz tego powinna być zapamiętywana i odtwarzana podczas ich wywoływania i nie może być wykorzystana do żadnego specjalnego zastosowania przez funkcje interfejsu binarnego aplikacji zdefiniowane dla architektury lub systemu operacyjnego. Kompilator wygeneruje ostrzeżenie, jeśli określony rejestr nie będzie odpowiedni. Gdy rejestr jest właściwy (użycie rejestru `ebx` wykorzystanego w powyższym przykładzie jest poprawne dla architektury x86), wówczas kompilator przestanie go wewnętrznie używać.

Taka optymalizacja może zapewnić duży przyrost wydajności, jeśli dana zmienna jest często wykorzystywana. Dobrym przykładem jej użycia jest maszyna wirtualna. Umieszczenie zmiennej w rejestrze, która przykładowo przechowuje wskaźnik stosu wirtualnego, może przyczynić się do znaczącego wzrostu wydajności. Z drugiej strony, jeśli dana architektura nie posiada zbyt wielu wolnych rejestrów (jak ma to miejsce w przypadku x86), wówczas przeprowadzanie takiej optymalizacji nie ma sensu.

Globalne zmienne rejestrowe nie mogą być używane w procedurach obsługi sygnałów lub przez więcej niż jeden wątek wykonawczy. Nie mogą również posiadać wartości początkowych, ponieważ w plikach wykonywalnych nie istnieje żaden mechanizm, który umożliwiałby dostarczanie domyślnych wartości do rejestrów. Deklaracje globalnych zmiennych rejestrowych powinny znajdować się przed wszystkimi definicjami funkcji.

Optymalizacja gałęzi kodu

Kompilator GCC pozwala programistom na przypisywanie oczekiwanej wartości wyrażenia — na przykład, aby przekazać mu informację, czy instrukcja warunkowa będzie prawdziwa lub fałszywa. Z kolei GCC może następnie wykonać operację porządkowania bloków oraz optymalizacje pozwalające na poprawę wydajności wykonania rozgałęzień warunkowych.

Składnia kompilatora GCC dla zdefiniowania optymalizacji gałęzi kodu jest wyjątkowo nieprzyjemna. Aby uczynić ją bardziej przyjazną, używane są następujące makra preprocesora:

```
#define likely(x)    __builtin_expect (!!(x), 1)
#define unlikely(x)  __builtin_expect (!!(x), 0)
```

Programiści mogą oznaczać wyrażenie, którego rezultat będzie z dużym prawdopodobieństwem prawdą lub fałszem, przez umieszczanie go odpowiednio w makrach `likely()` lub `unlikely()`.

Poniższy przykład oznacza gałąź kodu, którego wykonanie jest mało prawdopodobne (czyli odpowiedni wynik wyrażenia warunkowego będzie fałszem):

```
int ret;

ret = close (fd);
if (unlikely (ret))
   perror ("close");
```

I odwrotnie, następujący przykład oznacza gałąź kodu, którego wykonanie jest wysoko prawdopodobne:

```
const char *home;

home = getenv ("HOME");
if (likely (home))
   printf ("Twój katalog macierzysty jest równy %s\n", home);
else
   fprintf (stderr, "Zmienna środowiskowa HOME nie została ustawiona!\n");
```

Podobnie jak ma to miejsce w przypadku funkcji wplatanych, również w tym przypadku programiści często nadużywają optymalizacji gałęzi kodu. Podczas oznaczania wyrażeń może pojawić się pokusa, aby wykonać tę optymalizację dla *wszystkich* gałęzi kodu. Należy jednak postępować ostrożnie — gałęzie powinny być oznaczane jako prawdopodobne lub nieprawdopodobne jedynie wówczas, gdy jest się *niemal pewnym*, że wyrażenia będą zwracać prawdę lub fałsz *w prawie wszystkich przypadkach* (np., z prawdopodobieństwem równym 99%). Rzadko występujące błędy są dobrymi kandydatami do objęcia ich działaniem funkcji `unlikely()`. Należy jednak pamiętać, że niewłaściwa prognoza jest gorsza niż jej brak.

Uzyskiwanie typu dla wyrażenia

Kompilator GCC dostarcza słowa kluczowego `typeof()`, które pozwala na uzyskanie typu dla danego wyrażenia. Z punktu widzenia semantyki słowo kluczowe funkcjonuje w taki sam sposób jak `sizeof()`. Na przykład, poniższe wyrażenie zwraca typ dowolnej danej, która wskazywana jest przez wskaźnik `x`:

```
typeof (*x)
```

Wyrażenia tego można użyć, aby zadeklarować tablicę `y`, posiadającą otrzymany typ:

```
typeof (*x) y[42];
```

Popularnym zastosowaniem słowa kluczowego `typeof()` jest tworzenie „bezpiecznych" makr, w których można stosować dowolne wartości arytmetyczne i jednokrotnie obliczać ich parametry wejściowe:

```
#define max(a,b)        \
({                      \
   typeof (a) _a = (a); \
   typeof (b) _b = (b); \
   _a > _b ? _a : _b;   \
})
```

Uzyskiwanie wielkości wyrównania dla danego typu

Kompilator GCC dostarcza słowa kluczowego `__alignof__`, które pozwala na uzyskanie wielkości wyrównania dla danego obiektu. Wartość ta jest specyficzna dla architektury oraz interfejsu binarnego aplikacji (ABI). Jeśli aktualna architektura nie posiada wymaganego wyrów-

nania, słowo kluczowe zwraca wyrównanie rekomendowane przez ABI. W przeciwnym razie zwracane jest minimalne wymagane wyrównanie.

Składnia `__alignof__` jest identyczna jak w przypadku `sizeof()`:

```
__alignof__(int)
```

W zależności od architektury powyższa instrukcja zwróci prawdopodobnie wartość 4, ponieważ 32-bitowe liczby całkowite są w zasadzie wyrównane do granicy adresu, równej wielokrotności czterech bajtów.

Słowo kluczowe `__alignof__` działa również dla l-wartości. W tym przypadku zwracana wartość jest minimalnym wyrównaniem dla typu pierwotnego, a nie rzeczywistym wyrównaniem dla określonej l-wartości. Jeśli minimalna wartość wyrównania została zmieniona poprzez użycie atrybutu `aligned` (omówionego wcześniej w podrozdziale Zwiększanie wartości wyrównania dla zmiennej), wówczas zmiana ta zostaje odzwierciedlona podczas użycia `__alignof__`.

Przeanalizujmy poniższą strukturę:

```
struct ship
{
   int year_built;
   char canons;
   int mast_height;
};
```

Tę strukturę należy użyć w poniższym fragmencie kodu:

```
struct ship my_ship;

printf ("%d\n", __alignof__(my_ship.canons));
```

Użycie słowa kluczowego `__alignof__` w tym fragmencie kodu zwróci wartość `1`, mimo iż istnienie uzupełnienia w strukturze powoduje, że pole `canons` zużywa w rzeczywistości cztery bajty pamięci.

Pozycja elementu w strukturze

Kompilator GCC dostarcza wbudowanego słowa kluczowego pozwalającego na uzyskanie pozycji pola wewnątrz struktury. Makro `offsetof()`, zdefiniowane w pliku nagłówkowym `<stddef.h>`, jest fragmentem standardu ISO C. Większość definicji została zapisana w sposób wyjątkowo nieprzyjazny, przy użyciu niełatwej arytmetyki wskaźników oraz kodu, którego czytanie powinno być zakazane dla nieletnich programistów. Rozszerzenie GCC jest prostsze i potencjalnie szybsze:

```
#define offsetof(type, member) __builtin_offsetof (type, member)
```

Funkcja zwraca pozycję elementu `member` w strukturze `type`, to znaczy liczbę bajtów od początku tej struktury do danego elementu. Liczba ta może być również równa `0`. Weźmy pod uwagę następującą strukturę:

```
struct rowboat
{
   char *boat_name;
   unsigned int nr_oars;
   short length;
};
```

Rzeczywista pozycja zależy od rozmiaru zmiennych oraz wymagań wyrównania dla danej architektury i sposobu działania uzupełnienia, lecz w przypadku maszyny 32-bitowej należy oczekiwać, że wywołanie makra `offsetof()` w przypadku struktury `rowboat` oraz jej pól `boat_name`, `nr_oars` i `length` zwróci wartości równe odpowiednio `0`, `4` i `8`.

W systemie Linux makro `offsetof()` powinno być sformułowane przy użyciu słowa kluczowego kompilatora GCC i nie musi być przedefiniowane.

Uzyskiwanie powrotnego adresu funkcji

Kompilator GCC dostarcza słowa kluczowego pozwalającego na uzyskanie powrotnego adresu dla aktualnej funkcji lub dla jednej z procedur, które wywołują tę funkcję:

```
void * __builtin_return_address (unsigned int level)
```

Parametr `level` określa funkcję w łańcuchu wywołań, której adres powinien zostać zwrócony. Wartość `0` pozwala uzyskać powrotny adres dla aktualnej funkcji, wartość `1` pozwala na uzyskanie powrotnego adresu dla procedury wywołującej aktualną funkcję, wartość `2` umożliwia uzyskanie powrotnego adresu dla procedury wywołującej *poprzednią* procedurę i tak dalej.

Gdy aktualna funkcja jest wplatana, wówczas zwracanym adresem jest adres powrotny funkcji wywołującej. Jeśli jest to zachowanie niepożądane, należy użyć słowa kluczowego `noinline` (omówionego wcześniej w podrozdziale Zapobieganie wplataniu funkcji), aby zmusić kompilator, by nie wplatał danej funkcji.

Istnieje kilka zastosowań dla słowa kluczowego `__builtin_return_address`. Można go używać podczas wspomagania uruchamiania programu lub dla celów informacyjnych. Innym zastosowaniem jest rozwinięcie łańcucha wywołań w celu zaimplementowania introspekcji, narzędzia zrzutu systemowego, debuggera itd.

Należy zwrócić uwagę na to, że w przypadku niektórych architektur można uzyskać jedynie adres funkcji wywołującej. Użycie parametru niezerowego w takich architekturach może spowodować zwrócenie dowolnej wartości powrotnej. Dlatego też każdy parametr różny od zera jest nieprzenośny i powinien być używany jedynie dla celów związanych z uruchamianiem programu.

Zakresy funkcji case

Kompilator GCC zezwala na określanie zakresu wartości dla pojedynczego bloku w polu wyrażenia stałego instrukcji `case`. Ogólna składnia jest następująca:

```
case low ... high:
```

Na przykład:

```
switch (val)
{
   case 1 ... 10:
      /*... */
      break;
   case 11 ... 20:
      /*... */
      break;
```

```
    default:
        /* ... */
}
```

Funkcjonalność ta jest przydatna podczas użycia zakresu kodów ASCII:

```
case 'A' ... 'Z':
```

Należy zwrócić uwagę na to, że przed i za wielokropkiem powinny być wstawione spacje. Ich brak może spowodować błędne działanie kompilatora, szczególnie podczas użycia zakresów liczb całkowitych. Należy więc zawsze zapisywać zakres w taki sposób:

```
case 4 ... 8:
```

Nigdy:

```
case 4...8:
```

Arytmetyka wskaźników do funkcji oraz wskaźników void

Kompilator GCC zezwala na operacje dodawania i odejmowania dla wskaźników typu `void` oraz wskaźników do funkcji. Standardowo, ISO C nie zezwala na przeprowadzanie takich operacji, ponieważ określanie rozmiaru dla obiektu „pustego" (ang. *void*) jest działaniem niemającym sensu, gdyż zależy od tego, na co aktualnie wskazuje wskaźnik. Aby ułatwić wykonywanie takiej arytmetyki, w kompilatorze GCC założono, że rozmiar obiektu, do którego następuje odwołanie, równy jest jednemu bajtowi. Dlatego też poniższy fragment kodu powoduje zwiększenie wskaźnika `a` o wartość `1`:

```
a++; /* a jest wskaźnikiem void */
```

Użycie opcji `-Wpointer-arith` powoduje, że kompilator GCC wygeneruje ostrzeżenie, gdy takie rozszerzenia zostaną wykorzystane.

Więcej przenośności i elegancji za jednym razem

Faktem jest, że składnia `__attribute__` nie jest zbyt elegancka. Dla niektórych z rozszerzeń, przedstawionych w tym rozdziale, niezbędne jest ich zdefiniowanie za pomocą makr preprocesora, co ułatwi ich użycie. Bez względu na to, przy użyciu preprocesora czytelność wszystkich rozszerzeń może ulec dużej poprawie.

Nie jest to trudne do zrealizowania dzięki wykorzystaniu możliwości preprocesora. Oprócz tego, można jednocześnie uczynić rozszerzenia przenośnymi dzięki zdefiniowaniu ich w sposób niezależny od rodzaju użytego kompilatora, który nie musi być kompatybilny z GCC.

Aby to zrobić, należy umieścić poniższy fragment kodu w pliku nagłówkowym, który z kolei należy dołączyć do swoich plików źródłowych:

```
#if __GNUC__ >= 3
# undef inline
# define inline         inline __attribute__ ((always_inline))
# define __noinline     __attribute__ ((noinline))
# define __pure         __attribute__ ((pure))
# define __const        __attribute__ ((const))
# define __noreturn     __attribute__ ((noreturn))
```

```
# define __malloc       __attribute__ ((malloc))
# define __must_check   __attribute__ ((warn_unused_result))
# define __deprecated   __attribute__ ((deprecated))
# define __used         __attribute__ ((used))
# define __unused       __attribute__ ((unused))
# define __packed       __attribute__ ((packed))
# define __align(x)     __attribute__ ((aligned (x)))
# define __align_max    __attribute__ ((aligned))
# define likely(x)      __builtin_expect (!!(x), 1)
# define unlikely(x)    __builtin_expect (!!(x), 0)
#else
# define __noinline     /* brak rozszerzenia noinline */
# define __pure         /* brak rozszerzenia pure */
# define __const        /* brak rozszerzenia const */
# define __noreturn     /* brak rozszerzenia noreturn */
# define __malloc       /* brak rozszerzenia malloc */
# define __must_check   /* brak rozszerzenia warn_unused_result */
# define __deprecated   /* brak rozszerzenia deprecated */
# define __used         /* brak rozszerzenia used */
# define __unused       /* brak rozszerzenia unused */
# define __packed       /* brak rozszerzenia packed */
# define __align(x)     /* brak rozszerzenia aligned */
# define __align_max    /* brak rozszerzenia align_max */
# define likely(x)     (x)
# define unlikely(x)   (x)
#endif
```

Na przykład, za pomocą powyższego mechanizmu w następującym fragmencie kodu zdefiniowano funkcję czystą:

```
__pure int foo (void) { /*… */ }
```

Jeśli użyty zostanie kompilator GCC, funkcja otrzyma atrybut `pure`. Gdy kompilatorem nie będzie GCC, preprocesor zamieni element `__pure` na rozkaz pusty. Należy zwrócić uwagę, że przy pojedynczej definicji możliwe jest umieszczenie wielu atrybutów. Dzięki temu można dla takiej definicji bez problemu użyć więcej niż jednego makra preprocesora.

Oto rozwiązanie łatwiejsze, bardziej eleganckie i przenośne!

DODATEK B

Bibliografia

Poniższa bibliografia przedstawia zalecane pozycje książkowe, dotyczące programowania systemowego, podzielone na cztery podkategorie. Żadna z tych zalecanych prac nie musi zostać przeczytana. Jest to wyłącznie osobisty wybór autorski, wskazujący najbardziej przydatne książki dotyczące danego zagadnienia. Jeśli jednak zaistnieje potrzeba dokładniejszego zapoznania się z danym tematem, można wówczas skorzystać z przedstawionej bibliografii.

W przypadku niektórych z podanych pozycji książkowych zakłada się, że Czytelnik jest już biegły w określonym temacie, np. w programowaniu w języku C. Inne prace są rozległymi i dokładnymi dodatkami do tej książki, takimi jak pozycje dotyczące debuggera *gdb*, systemu kontroli wersji Subversion (*svn*) oraz projektowania systemu operacyjnego. Jeszcze inne zajmują się tematami, które nie zostały omówione w tym opracowaniu, takimi jak wielowątkowość gniazd. Bez względu na ich znajomość zalecane jest zapoznanie się ze wszystkimi, poniżej przedstawionymi pozycjami czytelniczymi. Oczywiście lista ta nie jest pełna — w razie potrzeb należy korzystać również z innych zasobów.

Programowanie w języku C

Poniżej przedstawione książki opisują programowanie w języku C, będącym uniwersalnym językiem programowania systemowego. Jeśli programowanie w języku C nie przychodzi tak łatwo jak posługiwanie się mową ojczystą, wówczas należy rozważyć zapoznanie się z jedną lub większą liczbą zaprezentowanych pozycji czytelniczych (jak również poświęcić wiele godzin na ćwiczenia praktyczne!). Warto poznać choćby pierwszą pracę, znaną powszechnie pod nazwą K&R. Jej zwięzłość pozwala na odkrycie prostoty języka C.

Brian W. Kernighan i Dennis M. Ritchie, *Język C*, WNT, Warszawa 1988.

> Książka ta, napisana przez twórcę języka C i jego współpracownika, jest biblią programowania w tym języku.

Peter Prinz and Tony Crawford, *C in a Nutshell*, O'Reilly Media, 2005.

> Wspaniała książka, omawiająca zarówno język C, jak również standardową bibliotekę języka C.

Peter Prinz i Ulla Kirch-Prinz, *C. Leksykon kieszonkowy*, Helion, Gliwice 2003.

> Zwięzłe kompendium wiedzy na temat języka C, umiejętnie uaktualnione dla standardu ANSI C99.

Peter van der Linden, *Expert C Programming*, Prentice Hall, 1994.

Wspaniała analiza mniej znanych aspektów programowania w języku C, przedstawiona przy użyciu fantastycznej inteligencji i poczucia humoru. Książka ta zawiera mnóstwo specyficznych żartów.

Steve Summit, *Programowanie w języku C. FAQ*, Helion, Gliwice 2003.

Ten hit wydawniczy zawiera ponad 400 często zadawanych pytań (razem z odpowiedziami na nie), dotyczących języka programowania C. Dla specjalistów języka C wiele z tych pytań i odpowiedzi jest oczywistych, lecz niektóre z trudniejszych powinny wywrzeć wrażenie nawet na najbardziej doświadczonych programistach języka C. Tylko prawdziwy mistrz języka C może odpowiedzieć na wszystkie pytania z tej książki! Jedyną jej wadą jest to, że nie uwzględnia ona standardu ANSI C99, który wprowadził jednak pewne zmiany. Należy zwrócić uwagę, że książka ta ma również wersję dostępną w sieci, która prawdopodobnie została całkiem niedawno uaktualniona.

Programowanie w Linuksie

Poniżej przedstawione pozycje wydawnicze omawiają programowanie w Linuksie, przeprowadzając jednocześnie analizę tematów nieuwzględnionych w tej książce (takich jak gniazda, IPC oraz *pthreads*), a także narzędzi wspomagających tworzenie oprogramowania (CVS, GNU Make, Subversion).

Richard W. Stevens, *Unix. Programowanie usług sieciowych t. 1 — API: gniazda i XTI*, WNT, Warszawa 2002

Tom ten poświęcony jest interfejsom programowym aplikacji, dotyczących gniazd. Niestety, nie omawia on programowania w Linuksie, lecz na szczęście uwzględnia standard IPv6.

Richard W. Stevens, *Unix. Programowanie usług sieciowych t. 2 — komunikacja międzyprocesowa*, WNT, Warszawa 2001

Znakomita analiza komunikacji międzyprocesowej (IPC).

Bradford Nichols et al., *PThreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly Media, 1996.

Przegląd interfejsów programowych aplikacji dotyczących obsługi wątków w standardzie POSIX, zwanych *pthreads*.

Robert Mecklenburg, *Managing Projects with GNU Make*, wyd. III, O'Reilly Media, 2004.

Doskonała analiza GNU Make — klasycznego narzędzia służącego do budowy projektów programistycznych w środowisku Linuksa.

Jennifer Versperman, *Essential CVS*, wyd. II, O'Reilly Media, 2006.

Doskonała analiza CVS — klasycznego narzędzia służącego do kontroli wersji i zarządzania kodem źródłowym w systemach uniksowych.

Ben Collins-Sussman et al., *Version Control with Subversion*, O'Reilly Media, 2004.

Wyjątkowa analiza systemu Subversion, będącego odpowiednim narzędziem do kontroli wersji oraz zarządzania kodem źródłowym w systemach uniksowych, przeprowadzona przez jego trzech twórców. Książka ta dostępna jest również całkowicie za darmo pod adresem *http://svnbook.red-bean.com*.

Arnold Robbins, *GDB Pocket Reference*, O'Reilly Media, 2005.

Podręczny leksykon wiedzy na temat *gdb* — debuggera, działającego w systemie Linux.

Ellen Siever et al, *Linux in a Nutshell*, wyd. V, O'Reilly Media, 2005.

Niesamowity podręcznik Linuksa, zawierający również opisy wielu narzędzi składających się na środowisko projektowe dla tego systemu.

Jądro Linuksa

Poniżej przedstawione dwie pozycje wydawnicze omawiają tematy związane z jądrem Linuksa. Istnieją trzy powody, aby zająć się tymi zagadnieniami. Po pierwsze, jądro udostępnia interfejs funkcji systemowych dla przestrzeni użytkownika i dlatego też jest „centrum" programowania systemowego. Po drugie, zachowanie i cechy charakterystyczne jądra rzucają światło na sposób współpracy z aplikacjami, które są przez niego uruchamiane. Po trzecie, jądro Linuksa zawiera wspaniały kod, a poniższe książki są napisane w interesujący sposób.

Robert Love, *Linux Kernel Development*, wyd. II, Novell Press, 2005.

Pozycja wydawnicza idealnie odpowiada tym programistom systemowym, którzy pragną wiedzieć, w jaki sposób projektuje się i implementuje jądro Linuksa. Nie jest to przegląd interfejsów API, lecz poważna analiza użytych algorytmów oraz decyzji, które są podejmowane przez jądro Linuksa.

Jonathan Corbet et al., *Linux Device Drivers*, wyd. III, O'Reilly Media, 2005.

Jest to wspaniały podręcznik omawiający tworzenie sterowników urządzeń dla jądra Linuksa, zawierający również doskonały opis interfejsów API. Analiza przeprowadzona w książce dotyczy sterowników urządzeń, jednakże nie zawiedzie ona również programistów o innych specjalnościach, w tym także programistów systemowych, którzy wnikliwie spoglądają w głąb mechanizmów jądra Linuksa. Bardzo dobry dodatek do poprzednio przedstawionej książki.

Projektowanie systemu operacyjnego

Poniżej przedstawione dwie pozycje nie dotyczą wyłącznie Linuksa, lecz omawiają w sposób abstrakcyjny projektowanie systemu operacyjnego. Jak zostało to już wielokrotnie wspomniane w tej książce, dokładne zrozumienie systemu, w którym tworzone jest oprogramowanie, pozwala na uzyskanie lepszych wyników.

Harvey Deitel et al, *Operating Systems*, wyd. III, Prentice Hall, 2003.

Dokładna prezentacja teorii dotyczącej projektowania systemu operacyjnego, połączona z pierwszorzędną analizą przypadków, prowadzącą od teorii do praktyki. To prawdopodobnie jedna z najlepszych pozycji, omawiających projektowanie systemu operacyjnego: jest nowoczesna, czytelna i kompletna.

Curt Schimmel, *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programming*, Addison-Wesley, 1994.

Mimo że książka ta nie jest bezpośrednio związana z programowaniem systemowym, oferuje ona jednak tak doskonały opis zagadnień, związanych ze współbieżnością oraz buforowaniem, że można śmiało rekomendować ją każdemu.

Skorowidz

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

Ś

T

U

V

W

X

Y

Z

O autorze

Robert Love jest od bardzo dawna hakerem oraz użytkownikiem systemu Linux. Interesuje się jądrem Linuksa, a także jest aktywnym uczestnikiem społeczności związanej z graficznym interfejsem użytkownika GNOME. Jego wkład w tworzenie jądra Linuksa zawiera między innymi prace związane z warstwą zdarzeń oraz interfejsem *inotify*. Udział Roberta Love w tworzeniu systemu GNOME dotyczy takich tematów jak Beagle, GNOME Volume Manager, NetworkManager oraz Project Utopia. Robert Love pracuje w firmie Google, w biurze programu Open Source (Open Source Program Office).

Jest autorem książki *Linux Kernel Development* (wydanej przez Novell Press). Obecnie ukazało się drugie wydanie tej pozycji. Robert Love jest również współautorem piątego wydania książki *Linux in a Nutshell* (opublikowanej przez wydawnictwo O'Reilly). Będąc aktywnym redaktorem czasopisma „Linux Journal", Love napisał wiele artykułów oraz wygłosił mnóstwo referatów dotyczących systemu Linux.

Robert Love ukończył University of Florida, gdzie uzyskał licencjat z matematyki oraz informatyki. Pochodzi z południowej Florydy. Obecnie mieszka w Bostonie.

Kolofon

Ilustracja na okładce książki *Programowanie systemowe w Linuksie* przedstawia człowieka w latającej maszynie. Zanim bracia Wright odbyli w 1903 roku swój pierwszy kontrolowany lot w maszynie cięższej od powietrza, ludzie na całym świecie dużo wcześniej próbowali latać przy użyciu zarówno prostych, jak i skomplikowanych maszyn. W drugim lub trzecim wieku Zhuge Liang z Chin podobno wzniósł się w powietrze przy użyciu lampionu Kongming — pierwszego balonu napełnionego gorącym powietrzem. W piątym lub szóstym wieku wielu Chińczyków rzekomo używało wielkich latawców, aby za ich pomocą odbywać loty powietrzne.

Chodzą także pogłoski, że stworzyli wirujące zabawki będące wczesnymi wersjami helikopterów. Ten projekt zainspirował Leonarda da Vinci. Słynny włoski malarz analizował lot ptaków i projektował spadochrony, a w 1485 roku zaprojektował ornitopter — maszynę latającą, naśladującą ruch skrzydeł ptaka, zdolną do przenoszenia ludzi. Mimo że nigdy nie została ona zbudowana, jednak jej ptakopodobna konstrukcja miała przez stulecia wpływ na sposób projektowania maszyn latających.

Konstrukcja przedstawiona na okładce jest bardziej skomplikowana od modelu maszyny szybującej, która nie posiadała śmigieł, skonstruowanej w 1893 roku przez Jamesa Meansa. Means wydrukował „podręcznik użytkownika" maszyny szybującej, w którym można odnaleźć stwierdzenie, iż „szczyt Mt. Willard, niedaleko Crawford House w New Hampshire, będzie doskonałym miejscem" do przeprowadzenia związanych z nią próbnych lotów.

Takie próby były bardzo niebezpieczne. W końcu dziewiętnastego wieku, Otto Lilienthal budował jednopłatowce, dwupłatowce oraz szybowce. Był pierwszym śmiałkiem, która udowodnił, że realizacja lotów leży w zasięgu możliwości człowieka. Po przeszło 2000 lotów szybowcowych na wysokości nawet ponad trzysta metrów, został nazwany „ojcem eksperymentów powietrznych". Zmarł w 1896 roku na skutek złamania kręgosłupa podczas lądowania awaryjnego.

Maszyny latające nazywane są mechanicznymi ptakami lub statkami powietrznymi. Czasami posiadają bardzo oryginalne nazwy, jak Mechaniczny Albatros. Zainteresowanie tematyką lotniczą nie maleje, a dzisiejsi entuzjaści aeronautyki rekonstruują dawne maszyny latające.

Ilustracja na okładce oraz grafika na początku każdego rozdziału pochodzą z archiwum ilustracji w Dover.