

Keith Haviland, Dina Gray, Ben Salama

UNIX

Programowanie systemowe



Programowanie()

Programowanie: Unix – programowanie systemowe

Keith Haviland, Dina Gray, Ben Salama

Tytuł oryginalu amerykańskiego: UNIX System Programming, Second edition

Copyright © Addison Wesley Longman Inc. 1999

This translation of UNIX System Programming, Second edition is published by arrangement with Pearson Education Limited.

Opracowanie wersji polskiej: Bogdan Kamiński

Copyright © for the Polish edition by Wydawnictwo RM

Warszawa 1999

Wydawnictwo RM, 00-987 Warszawa 4, skr. poczt. 144

e-mail: rm@rm.com.pl

WWW: http://www.rm.com.pl

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without prior permission in writing from the Publisher.

Żadna część tej pracy nie może być powielana i rozpowszechniana w jakiekolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny) włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów, bez wcześniejszej pisemnej zgody wydawcy.

Wszystkie nazwy handlowe i towarów, występujące w niniejszej publikacji, są znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicielami.

Printed in Poland.

Wydawnictwo RM dołożyło wszelkich starań, aby zapewnić najwyższą jakość tej książce. Jednakże nikomu nie udziela żadnej rękojmi ani gwarancji. Wydawnictwo RM nie jest w żadnym wypadku odpowiedzialne za jakiekolwiek szkody łącznie ze szkodami z tytułu utraty zysków związanych z prowadzeniem przedsiębiorstwa, przerw w działalności przedsiębiorstwa lub utraty informacji gospodarczej, będących następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeżeli Wydawnictwo RM zostało zawiadomione o możliwości wystąpienia szkód.

ISBN 83-7243-016-0

Redaktor prowadzący: Krystyna Knap

Redakcja: Mirosława Szymańska

Projekt graficzny okładki: Grażyna Jędrzejec

Korekta: Jolanta Konieczna

Skład: Magdalena Mocarska

Druk i oprawa: Oficyna Wydawnicza READ ME – Drukarnia w Łodzi

Wydanie I

10 9 8 7 6 5 4 3 2 1

S.57072/3

Niall, nieznacznie młodszej od
wydania pierwszego

KH

Zoe, która osiągnęła tak dużo
w tak krótkim życiu

DG

PRZEDMOWA

Koncepcja książki

System operacyjny Unix miał swoje skromne początki w Bell Laboratories w 1969 roku, zyskując następnie z każdym dniem na znaczeniu i popularności. Najpierw przypadł do gustu środowiskom akademickim, ale już w latach osiemdziesiątych stał się rzeczywistym standardem systemu operacyjnego nowej generacji mikro - i minikomputerów dla wielu użytkowników. W czasie, gdy powstaje ta książka, jego rozwój trwa nadal.

Tak zaczynało się pierwsze wydanie tej książki z roku 1987¹. Teraz, po upływie ponad dziesięciu lat, okazało się, że Unix spełnił pokładane w nim nadzieje, będąc kluczową częścią technologii, którą przeniesiemy do następnego stulecia. Pozycja Uniksa w środowiskach naukowych i technicznych jest tradycyjnie mocna; obecnie platformę Uniksa wykorzystuje również wiele systemów przetwarzania transakcji i zarządzania danymi. Stanowi on oczywiście także jądro podstawowych serwerów Internetu.

Książka *Unix – programowanie systemowe* również została dobrze przyjęta. Do dziś ukazało się wiele wznowień jej pierwszego wydania, pojawiających się raz lub nawet dwa razy do roku. Sprawiła to nadzwyczajna aktualność tej książki, odzwierciedlającej standard i trwaną naturę systemu operacyjnego Unix. Teraz nadszedł czas na drugie wydanie i wprowadzenie poważniejszych zmian. Chcieliśmy zatrudnić w książce także tematy odnoszące się do bardziej rozproszonego środowiska typowych rozwiązań informatycznych końca lat dziewięćdziesiątych.

Jednak koncepcja książki pozostała niezmieniona. Ponownie koncentrujemy się na interfejsie programowym pomiędzy jądrem (ang. *kernel*) Uniksa (czyli częścią Uniksa, która jest właściwym systemem operacyjnym) a programami aplikacji, działającymi w środowisku Uniksa. Jest on często nazywany interfejsem funkcji systemowych (ang. *system call interface*) Uniksa (choć różnica między tymi funkcjami i zwykłymi procedurami bibliotecnymi nie jest tak jasna, jak być powinna). Przekonamy się, że takie funkcje są podstawowymi składnikami, z których zbudowano wszystkie programy uniksowe – zarówno dostarczane razem z systemem operacyjnym, jak też opracowane niezależnie. Nasza książka przeznaczona jest dla programistów mających już pewne doświadczenie w Uniksie, którzy będą rozwijać oprogramowanie uniksowe w języku C. Treść książki powinna być także istotna dla osób opracowujących programy na poziomie systemu operacyjnego oraz programy aplikacji i biznesowe – dla każdego rzeczywiście zainteresowanego rozwojem programów uniksowych.

1. Chodzi o pierwsze wydanie amerykańskie (przyp. red.)

Poza funkcjami systemowymi omówimy też ważniejsze biblioteki procedur, dostarczane razem z systemem Unix. Są one oczywiście napisane z wykorzystaniem funkcji systemowych i w wielu przypadkach wykonują podobne działania, ale na wyższym poziomie lub w bardziej przyjazny sposób. Wyjaśniając funkcje systemowe i biblioteki procedur mamy nadzieję, że dzięki temu nie będziesz wynajdować ponownie kola i lepiej zrozumiesz wewnętrzne działanie tego wciąż eleganckiego systemu operacyjnego.

Specyfikacja X/Open

Unix ma długą historię. W tym czasie obróśł w wiele formalnych i rzeczywistych standardów, powstały jego odmiany handlowe i akademickie. Jednak jądro systemu – na którym koncentruje się ta książka – pozostało niezmienione.

Pierwsze wydanie książki *Unix – programowanie systemowe* było oparte na Issue 2 (wydaniu 2) *System V Interface Definition* (SVID) firmy AT&T. System V jest nadal jedną z najlepszych praktycznych realizacji oprogramowania Uniksa. W drugim wydaniu w większości oparliśmy tekst i przykłady na publikacji *X/Open Issue 4 Version 2: System Interface Definition i System Interfaces and Headers* oraz fragmentach publikacji *Networking Services* – pochodzących z 1994 roku. Dla wygody będziemy nazywali ten zestaw dokumentów *XSI*, co stanowi skrót od *X/Open System Interface*. Zauważ, że gdy to konieczne, omawiamy cechy rzeczywistych realizacji systemu.

Przyda się tu trochę dodatkowych informacji. X/Open powstał jako konsorcjum dostawców sprzętu (interesujących się systemami otwartymi i Uniksem jako platformą), które z czasem znacznie się rozrosło. Jedno z głównych zadań tego konsorcjum stanowiła praktyczna standaryzacja Uniksa, a jego podstawowy podręcznik przenośności (nazywany *XPG*) stosowało jako punkt wyjścia kilku większych dostawców (włącznie z Sunem, IBM, Hewlett-Packardem, Novelliem i Open Software Foundation), znanych jako **Spec 1170 Initiative**. (1170 odnosi się do liczby wywołań funkcji, nagłówków, poleceń i narzędzi zawartych w ćwiczeniach!) Celem konsorcjum było opracowanie pojedynczej, połączonej specyfikacji usług systemowych Uniksa, włącznie z funkcjami systemowymi, które są omawiane w tej książce. W efekcie powstał ogólny zestaw praktycznych specyfikacji, łączących wiele z kolidujących ze sobą strumieni standardów Uniksa, których główną część stanowią wymienione powyżej kluczowe dokumenty. Pozostałe opracowane dokumenty zawierają polecenia Uniksa i obsługę ekranu.

Z perspektywy programowania systemowego dokumenty *XSI* tworzą pragmatyczny punkt wyjścia; przykłady z tej książki powinny działać na większości aktualnych platform Uniksa. Standard X/Open pociągnął za sobą kolejne związane i uzupełniające standardy oraz praktykę. Pochodzi on od standardów C ANSI/ISO, ważnego podstawowego standardu *POSIX* (IEEE Std 1003.1-1990), SVID, elementów specyfikacji Open Software Foundation i najczęściej używanych procedur z bardzo wpływowego Uniksa Berkeley.

Oczywiście normalizacja jest procesem ciągłym. X/Open i OSF (Open Software Foundation) połączyły się w 1996 roku w **The Open Group**. Najnowsze oprawa-

cowania *POSIX* (do których sięgaliśmy w trakcie pisania książki) potwierdzają wrażenie, że The Open Group odnosi się do Version 2 (wersji 2) *Single UNIX Specification*, która z kolei zawiera Issue 5 (wydanie 5) *System Interface Definition, System Interfaces and Headers i Networking Services*. Ważne, ale specjalistyczne rozszerzenia obejmują takie obszary jak wątki (ang. *threads*), rozszerzenia czasu rzeczywistego i łączenie dynamiczne.

Na koniec zwróć uwagę, że:

1. Każdy standard, zawierający alternatywne sposoby wykonywania różnych rzeczy i rzadko wykorzystywanych, ale ciągle ważną funkcjonalność, jest bardzo obszerny. Skoncentrujemy się więc na tym, co da ci dobre podstawy programowania w Uniksie. Materiał ten nie obejmuje jednak całej zawartości standardów.
2. Jeśli ważne jest dla ciebie spełnienie formalnych standardów, powinieneś rozważyć umieszczenie (i wypróbowanie) w swoim programie odpowiednich znaczników, jak `_XOPEN_SOURCE` lub `_POSIX_SOURCE`.

Kompozycja książki

Materiał w książce został rozłożony następująco:

- **Rozdział 1** zawiera przegląd kilku zasadniczych pojęć i podstawowej terminologii. Dwa najważniejsze omawiane w rozdziale terminy to **plik** (ang. *file*) i **proces** (ang. *process*). Mamy nadzieję, że większość czytelników książki znana jest przynajmniej częścią przedstawionego tu materiału (sprawdź warunki wstępne opisane w kolejnej części).
- **Rozdział 2** opisuje podstawowe funkcje systemowe do obsługi plików, takie jak: otwieranie i tworzenie plików, czytanie danych oraz plików, a także zapis do plików i bezpośredni dostęp (ang. *random access*). Wprowadzamy sposoby obsługi błędów, które mogą być generowane przez funkcje systemowe.
- **Rozdział 3** traktuje pliki całościowo. Przyjrzymy się tu prawom własności pliku, zarządzaniu przez Unix przywilejami systemu plików oraz zmianie tych atrybutów za pomocą funkcji systemowych.
- **Rozdział 4** omawia głównie pojęcie **katalogu** (ang. *directory*) w Uniksie z punktu widzenia programowania. Włączyliśmy tu krótkie omówienie podstawowej struktury przechowywania plików w Uniksie, wraz z **systemem plików** (ang. *file systems*) i **plikami specjalnymi** (ang. *special files*) używanymi do reprezentacji urządzeń.
- **Rozdział 5** opisuje podstawy natury i sterowania procesami Uniksa. Zostały tu szczegółowo objaszone ważne funkcje systemowe: `fork` i `exec`. Materiał przykładowy stanowi m.in. mala powłoka (ang. *shell*) lub procesor poleceń (ang. *command processor*).
- **Rozdział 6** jest pierwszym z trzech rozdziałów poświęconych komunikacji międzyprocesowej (ang. *inter-process communication, IPC*). Przedstawia on

- **sygnały** (ang. *signals*) i **obsługę sygnałów** (ang. *signal handling*), użyteczne przy wychwytywaniu i komunikowaniu o nieprawidłowych warunkach.
- **Rozdział 7** omawia **potok** (ang. *pipe*), najbardziej użyteczną technikę komunikacji międzyprocesowej Uniksa, w którym wyjście jednego programu może być podłączone do wejścia innego. Rozważamy tworzenie, czytanie i pisanie z zastosowaniem potoków oraz wybór z wielu potoków.
- **Rozdział 8** zawiera techniki komunikacji międzyprocesowej, wprowadzone do Uniksa przez System V. Opisano tu: **blokowanie rekordu** (ang. *record locking*), **przekazywanie komunikatu** (ang. *message passing*), **semafony** (ang. *semaphores*) i **wspólną pamięć** (ang. *shared memory*).
- **Rozdział 9** opisuje pracę z terminaliem na poziomie funkcji systemowych. Pojawia się tu przykład wykorzystania **pseudoterminali**.
- **Rozdział 10** omawia pokrótkę pracę w sieci Uniksa i sposób wykorzystania **gniazd** (ang. *sockets*) do przesyłania komunikatów z jednego komputera do innego.
- W **rozdziale 11** porzucamy funkcje systemowe i zaczynamy rozważania na temat głównych pakietów bibliotek. Znajdziemy tu systematyczne omówienie **Standardowej Biblioteki I/O** (ang. *Standard I/O Library*), zawierającej znacznie więcej możliwości obsługi plików niż podstawowe funkcje systemowe wprowadzone w rozdziale 2.
- **Końcowy rozdział 12** jest przeglądem różnych funkcji systemowych i procedur bibliotecznych, ponieważ wiele z nich jest bardzo istotnych przy tworzeniu rzeczywistych programów. Wśród omawianych tematów znajdują się: obsługa napisów, funkcje związane z czasem i zarządzanie pamięcią.

Co powinieneś wiedzieć

Nasza książka nie ma na celu ogólnego wprowadzenia do Uniksa lub języka C, ale stanowi szczegółowe omówienie interfejsu funkcji systemowych Uniksa. Wyniesiesz z niej największe korzyści, jeśli zaznajomisz się wcześniej z następującymi tematami:

- rejestrowanie się w systemie Unix;
- tworzenie plików za pomocą jednego z dostępnych w systemie standarodowych edytorów;
- drzewiasta struktura katalogów Uniksa;
- podstawowe polecenia manipulacji plikami i katalogami;
- tworzenie i komplikacja prostych programów w C (włącznie z programami, których kod zawarty jest w wielu plikach źródłowych);
- proste wykorzystanie w programie w C procedur I/O: `printf` i `getchar`;
- używanie w programie w C argumentów wiersza poleceń, to jest `argc` i `argv`;

- używanie podręcznika systemowego (ang. *system's manual*). (Niestety, nie jest już możliwe dawanie konkretnych porad w tym zakresie, ponieważ początkowy standardowy format podręcznika został zmodyfikowany przez kilku twórców. Tradycyjny podręcznik jest podzielony na osiem części, uporządkowanych alfabetycznie. Dla nas najbardziej interesujące będą pierwsze trzy: *Section 1* opisuje polecenia, *Section 2* – funkcje systemowe, a *Section 3* – podprogramy).

Jeśli nie czujesz się najlepiej w którymś z powyższych tematów, powinieneś wykonać wymienione poniżej ćwiczenia. Jeżeli potrzebujesz więcej pomocy, bibliografia zamieszczona na końcu książki wskaże ci odpowiedni tekst.

Jeszcze jedna uwaga na zakończenie: technika komputerowa *nie jest* widowiskowym sportem – dlatego też w książce tej kładziemy silny nacisk na ćwiczenia i przykłady. Zanim zaczniesz, powinieneś zapewnić sobie dostęp do odpowiedniego komputera uniksowego.

Ćwiczenie P.1 Wyjaśnij zastosowanie następujących poleceń Uniksa:

```
ls cat rm cp mv mkdir cc
```

Ćwiczenie P.2 Stosując swój ulubiony edytor utwórz mały plik tekstowy. Użyj `cat` do utworzenia innego pliku, który zawiera pięć powtórzeń zawartości tego pliku. Używając `wc` policz ilość znaków i słów w obu plikach. Wyjaśnij wyniki. Utwórz podkatalog i przenieś do niego oba pliki.

Ćwiczenie P.3 Utwórz plik zawierający wydruk zawartości twojego katalogu macierzystego oraz katalogu `/bin`.

Ćwiczenie P.4 Wymyśl pojedynczy wiersz polecień, który pokaże liczbę użytkowników aktualnie zarejestrowanych w systemie.

Ćwiczenie P.5 Napisz, skompiluj i wykonaj program w C, który wydrukuje wybrany przez ciebie komunikat powitalny.

Ćwiczenie P.6 Napisz, skompiluj i uruchom program w C, który wydrukuje swoje argumenty.

Ćwiczenie P.7 Używając funkcji `getchar()` i `printf()` napisz program, który liczy ilość słów, wierszy i znaków na swoim wejściu.

Ćwiczenie P.8 Utwórz plik zawierający podprogram w C, który drukuje komunikat 'hello, world'. Utwórz oddzielnny plik zawierający główny program, który wywołuje tę procedurę. Skompiluj i wykonaj program wynikowy, nazywając go `hw`.

Ćwiczenie P.9 Znajdź w podręczniku swojego systemu pozycje dla następujących tematów: polecenie `cat`, podprogram `printf` i funkcja systemowa `write`.

dziękowania

Chcielibyśmy podziękować Steve'owi Pate, Nigelowi Barnesowi, Andrisowi Storsowi, Jasonowi Reedowi, Philowi Tomkinsowi, Colinowi White, Joe'owi i Victorii Cave. Chcielibyśmy także podziękować Dylanowi Reisengerowi, Steve'owi Temblettowi i Karen Mosman za ich pomoc podczas tworzenia wydania drugiego.

winniśmy także podziękować osobom, które pomagały przy wydaniu pierwszym: Steve'owi Ratcliffe'owi za czytanie wielu wersji każdego rozdziału w pierwszym wydaniu i sprawdzaniu wszystkich *oryginalnych* przykładowych programów; a także Jonathanowi Lefflerowi, Gregowi Broughamowi, Dominice Hoplo, Nigelowi Martinowi, Billowi Fraser-Campbellowi, Dave'owi Lukesowi i ydowi Williamsowi za ich komentarze, sugestie oraz pomoc podczas przygotowywania tekstu.

Spis treści

Przedmowa	VII
Rozdział 1: Podstawowe pojęcia i terminologia	1
1.1 Plik	1
1.2 Proces	4
1.3 Funkcje systemowe i procedury biblioteczne	4
Rozdział 2: Plik	7
2.1 Elementarne operacje dostępu do pliku	7
2.2 Standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach	28
2.3 Przegląd Standardowej Biblioteki I/O	32
2.4 Zmienna errno i funkcje systemowe	35
Rozdział 3: Plik w kontekście	37
3.1 Pliki w środowisku wielu użytkowników	37
3.2 Pliki z wieloma nazwami	47
3.3 Uzyskiwanie informacji o pliku: stat i fstat	50
Rozdział 4: Katalogi, systemy plików i pliki specjalne	57
4.1 Wstęp	57
4.2 Katalogi od strony użytkownika	58
4.3 Implementacja katalogu	60
4.4 Programowanie i katalogi	64
4.5 Systemy plików Uniksa	74
4.6 Pliki urządzeń Uniksa	77
Rozdział 5: Proces	83
5.1 Pojęcie procesu – raz jeszcze	83
5.2 Tworzenie procesów	84
5.3 Uruchamianie nowych programów za pomocą exec	87
5.4 Łączne użycie exec i fork	92
5.5 Dziedziczenie danych i deskryptorów plików	95
5.6 Koniec procesów za pomocą funkcji systemowej exit	97
5.7 Synchronizacja procesów	99
5.8 Przedwczesne zakończenia i procesy zombie	102
5.9 Procesor poleceń: smallsh	103
5.10 Atrybuty procesu	109

Rozdział 6: Sygnały i przetwarzanie sygnałów	119
6.1 Wprowadzenie	119
6.2 Obsługa sygnałów	126
6.3 Blokowanie sygnałów	135
6.4 Wysyłanie sygnałów	136
Rozdział 7: Komunikacja międzyprocesowa za pomocą potoków	145
7.1 Potoki	145
7.2 Pliki FIFO lub nazwane potoki	165
Rozdział 8: Zaawansowana komunikacja międzyprocesowa	171
8.1 Wprowadzenie	171
8.2 Blokowanie rekordu	172
8.3 Zaawansowane urządzenia IPC	181
Rozdział 9: Terminal	209
9.1 Wprowadzenie	209
9.2 Terminal uniksowy	212
9.3 Spojrzenie ze strony programu	217
9.4 Pseudoterminale	232
9.5 Przykład obsługi terminala: program tscript	236
Rozdział 10: Gniazda	243
10.1 Wprowadzenie	243
10.2 Rodzaje połączeń	244
10.3 Adresowanie	245
10.4 Interfejs gniazd	246
10.5 Programowanie modelu połączeniowego	247
10.6 Programowanie modelu bezpołączeniowego	257
10.7 Różnice między modelami	260
Rozdział 11: Standardowa Biblioteka I/O	261
11.1 Wprowadzenie	261
11.2 Struktura FILE	262
11.3 Otwieranie i zamykanie strumieni plików: fopen i fclose	263
11.4 Operacje I/O dla pojedynczego znaku: getc i putc	265
11.5 Zwracanie znaku do strumienia pliku: ungetc	267
11.6 Standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach	268
11.7 Standardowe procedury stanu I/O	270
11.8 Wejście i wyjście linii	271

11.9 Wejście i wyjście binarne: fread i fwrite	273
11.10 Bezpośredni dostęp do pliku: fseek, rewind i tell	276
11.11 Formatowane wyjście: rodzina funkcji printf	277
11.12 Formatowane wejście: rodzina funkcji scanf	283
11.13 Uruchamianie programów za pomocą Standardowej Biblioteki I/O	286
11.14 Rozmaite funkcje	292
Rozdział 12: Rozmaite funkcje systemowe i procedury biblioteczne	295
12.1 Wprowadzenie	295
12.2 Dynamiczne zarządzanie pamięcią	295
12.3 I/O odwzorowane w pamięci i manipulacja pamięcią	302
12.4 Czas	306
12.5 Napisy i manipulacja znakami	308
12.6 Inne przydatne funkcje	312
Dodatek A: Kody błędów errno i związane z nimi komunikaty	315
A.1 Wprowadzenie	315
A.2 Lista kodów błędów i komunikatów	316
Dodatek B: Główne standardy	323
B.1 Historia	323
B.2 Kluczowe standardy	324
Bibliografia	325
Indeks	327

ROZDZIAŁ 1

Podstawowe pojęcia i terminologia

- 1.1 Plik
- 1.2 Proces
- 1.3 Funkcje systemowe i procedury biblioteczne

Będzie to krótki przegląd kilku podstawowych pojęć i terminologii, używanych w tej książce. Zaczniemy od omówienia pojęcia plik w Uniksie.

1.1 Plik

Informacja w systemie Unix jest przechowywana w plikach. Oto typowe polecenia Uniksa, manipulujące plikami:

`$ vi my_test.c`

powinno wywołać edytor vi, by utworzyć lub edytować plik `my_test.c`,

`$ cat my_test.c`

powinno wyświetlić zawartość `my_test.c` na terminalu, a:

`$ cc -o my_test my_test.c`

powinno wywołać kompilator C w celu generacji pliku programu `my_test` z pliku źródłowego `my_test.c`, zakładając oczywiście, że `my_test.c` nie zawiera żadnych błędów składni.

Większość plików posiada pewien rodzaj struktury logicznej, найдanej przez użytkownika, który je utworzył. Na przykład dokument będzie składał się ze słów, linii, akapitów i stron. Jednak dla systemu wszystkie pliki uniksowe wyglądają jak proste, niestrukturalne sekwencje bajtów lub znaków. Dostarczane przez system pierwotne funkcje dostępu do pliku pozwalają na dostęp sekwencyjny lub swobodny do indywidualnych bajtów. W pliku nie ma wstawionych żadnych znaków kończących rekordy lub plik ani żadnych typów rekordów.

Ta prostota jest całkowicie zamierzona i typowa dla filozofii Uniksa. Plik uniksowy to czysta, ogólna koncepcja, na podstawie której mogą być opracowane bardziej złożone i ścisłe struktury (takie jak organizacja plików indeksowych). Zbędne

szczegóły i specjalne przypadki zostały bezlitośnie wyeliminowane. Na przykład znak nowego wiersza (w rzeczywistości znak ASCII wysunięcia wiersza) wewnętrz zwykłego pliku tekstowego, który wskazuje koniec linii tekstu, jest tak samo jak inne znaki czytany lub zapisywany przez narzędzia systemowe i programy użytkownika. Tylko programy oczekujące, że ich wejście składa się z poszczególnych linii, muszą zwracać uwagę na semantykę znaku nowego wiersza.

Unix nie odróżnia różnych rodzajów plików. Plik może zawierać czytelny tekst (taki jak lista zakupów albo akapit, który teraz czytasz) albo dane „binarne” (jak skompilowana postać programu). W obu przypadkach do manipulacji plikiem mogą być używane te same pierwotne operacje albo narzędzia. Wniosek z tego taki, że nie znajdziesz tu żadnego formalnego schematu nazw, jak w innych systemach operacyjnych (jednak niektóre programy, jak na przykład cc, stosują pewną prostą konwencję nazw). Nazwy plików uniksowych są całkowicie dowolne i w SVR4 (System V Release 4) mogą mieć długość do 255 znaków. Jednak XSI określa, że w celu zapewnienia prawdziwej przenośności nie powinny przekraczać 14 znaków długości – granicy istniejącej we wcześniejszych wersjach Uniksa.

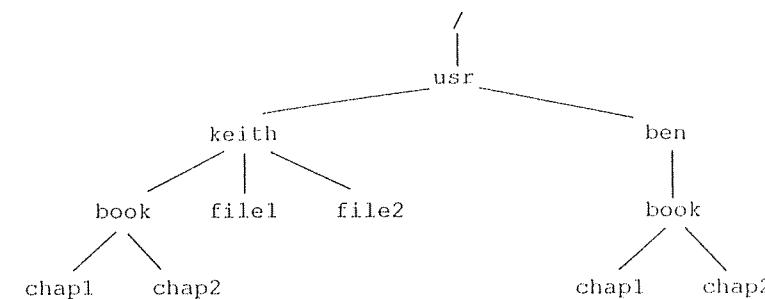
1.1.1 Katalogi i nazwy ścieżek

Ważnym pojęciem związanym z plikiem jest **katalog**. Katalogi są zbiorami plików i pozwalają na pewną logiczną organizację informacji zawartej w systemie. Na przykład każdy użytkownik zwykle ma własny katalog „macierzysty” do pracy, podczas gdy polecenia, biblioteki systemowe i programy administracyjne są zasadniczo umieszczane w ich własnych, określonych katalogach. Katalogi również dobrze mogą zawierać dowolną liczbę podkatalogów. Te z kolei mogą zawierać ich własne podkatalogi i tak dalej. W rzeczywistości katalogi mogą być zagnieżdżane do dowolnej głębokości. Dlatego pliki uniksowe są zorganizowane w hierarchiczną strukturę drzewiastą, gdzie każdy węzeł nie będący końcowym odpowiada katalogowi. Na szczytce tego drzewa znajduje się pojedynczy katalog, umownie nazywany **katalogiem głównym** (ang. *root directory*).

Strukturę katalogów Uniksa omawiamy szczegółowo w rozdziale 4. Jednak ponieważ w tekście będziemy używać plików uniksowych, warto zauważyć, że pełne ich nazwy – określone jako **nazwy ścieżek** (ang. *pathnames*) – odzwierciedlają strukturę drzewa. Każda nazwa ścieżki podaje sekwencję katalogów, prowadzących do pliku. Na przykład nazwa ścieżki:

`/usr/keith/book/chap1`

może być rozłożona następująco: pierwszy znak / oznacza, że nazwa ścieżki zaczyna się w katalogu głównym; tak więc ta nazwa ścieżki podaje *bezwzględne* położenie pliku wewnętrz pamięci plików. Następnie znajduje się `usr`, będący podkatalogiem katalogu głównego. Katalog `keith` jest kolejnym krokiem w dół drzewa, a więc podkatalogiem `/usr`. Podobnie katalog `book` jest podkatalogiem `/usr/keith`. Końcowy składnik, `chap1` również dobrze mógłby być katalogiem, a nie zwykłym plikiem, ponieważ katalogi są identyfikowane przez taki sam układ nazw, jak pliki. Przykład drzewa katalogu zawierającego tę nazwę ścieżki pokazuje rysunek 1.1.



Rysunek 1.1 Przykład drzewa katalogu

Nazwa ścieżki, która nie zaczyna się znakiem / jest nazywana **względna nazwą ścieżki** (ang. *relative pathname*) i podaje drogę do pliku względem bieżącego katalogu roboczego (ang. *current working directory*) użytkownika. Na przykład nazwa ścieżki:

`chap1/intro.txt`

opisuje plik `intro.txt`, zawarty w podkatalogu `chap1` bieżącego katalogu. W szczególnym przypadku nazwa:

`intro.txt`

po prostu identyfikuje plik `intro.txt` w bieżącym katalogu roboczym. Ponownie zwróć uwagę, że dla programu, który ma być *naprawidł* przenośny, każdy indywidualny składnik nazwy ścieżki powinien być ograniczony do 14 znaków długości.

1.1.2 Właściwości i uprawnienia

Plik nie jest charakteryzowany wyłącznie przez zawarte w nim dane: istnieje pewna liczba innych podstawowych atrybutów skojarzonych z każdym plikiem uniksowym. Na przykład każdy plik jest **właściwością** konkretnego użytkownika. Relacja właściwości (ang. *ownership*) daje pewne prawa; jednym z nich jest możliwość zmiany innych rodzajów atrybutów pliku, mianowicie **uprawnień** (ang. *permissions*). Jak zobaczymy w rozdziale 3, uprawnienia decydują o tym, którzy użytkownicy mogą czytać lub zapisywać plik, albo też – jeżeli plik zawiera program – uruchomić go.

1.1.3 Uogólnienie pojęcia pliku

Unix rozszerza pojęcie pliku tak, że zawiera ono nie tylko zwykłe pliki (**prawidłowe pliki** (ang. *regular files*) w terminologii Uniksa), ale też urządzenia peryferyjne i kanały komunikacji międzyprocesowej. Oznacza to, że te same podstawowe operacje mogą być używane do czytania i zapisywania plików tekstowych oraz binarnych, a także terminali, jednostek taśm magnetycznych i nawet pamięci głównej. Umożliwia to programom być ogólnymi narzędziami, zdolnymi do używania każdego typu urządzenia. Na przykład:

```
$ cat file > /dev/rmt0
```

stanowi prymitywny sposób zapisu pliku na taśmie magnetycznej (nazwa ścieżki `/dev/rmt0` jest wspólną mnemoniką dla jednostki taśmy magnetycznej).

1.2 Proces

W terminologii Uniksa **proces** jest po prostu egzemplarzem wykonywanego programu. Najłatwiejszy sposób utworzenia procesu to wydanie polecenia procesorowi poleceń Uniksa lub **powłoce** (ang. *shell*). Na przykład jeśli użytkownik napisze:

```
$ ls
```

proces powłoki, który przyjmuje polecenie, powinien utworzyć inny proces specjalnie w celu uruchomienia programu wydruku zawartości katalogu `ls`. Ponieważ Unix jest systemem wielozadaniowym, jednocześnie może działać więcej niż jeden proces. W rzeczywistości powinien to być co najmniej jeden proces (zwykle jest ich więcej), dla każdego bieżącego użytkownika systemu.

1.2.1 Komunikacja międzyprocesowa

Unix pozwala współpracować procesom współbieżnym za pomocą rozmaitych metod komunikacji międzyprocesowej (ang. *inter-process communication, IPC*).

Jedną z takich metod jest **potok** (ang. *pipe*). Potoki są zwykle używane do łączenia wyjścia jednego programu z wejściem innego bez przechowywania danych w pliku pośredniczącym. I znów użytkownicy mogą skorzystać z tego ogólnego urządzenia za pomocą powłoki. Linia poleceń:

```
$ ls | wc -l
```

powoduje, że powłoka tworzy dwa współbieżne procesy: uruchamia `ls` i program liczenia słów `wc`. Łączy także wyjście `ls` z wejściem `wc`. Wynikiem jest liczba plików w bieżącym katalogu.

Inny środek komunikacji międzyprocesowej w Uniksie stanowią **sygnały** (ang. *signals*), oferujące model komunikacji oparty na przerwaniach. Bardziej zaawansowane środki to pamięć **wspólna** (ang. *shared memory*) i **semafory** (ang. *semaphores*). Także **gniazda** (ang. *sockets*), zwykle stosowane przy współpracy procesów przez sieć, mogą być używane do prostej IPC między procesami na tym samym komputerze.

1.3 Funkcje systemowe i procedury biblioteczne

W przedmowie powiedzieliśmy, że wiodącym tematem tej książki jest **interfejs funkcji systemowych** (ang. *system call interface*). Wydaje się, że termin **funkcje systemowe** wymaga dokładniejszego zdefiniowania.

Funkcje systemowe są w rzeczywistości paszportem programisty do **jądra** (ang. *kernel*) Uniksa. Jądro, o którym po raz pierwszy wspomnieliśmy w przedmowie, jest pojedynczym fragmentem programu, znajdującym się na stale w pamięci i mającym do czynienia z planowaniem procesów systemowych Uniksa oraz sterowaniem I/O. W istocie jądro jest tą częścią Uniksa, od której zależą właściwości systemu operacyjnego. Wszystkie procesy użytkowników i każdy dostęp do systemu plików powinny być udostępniane, monitorowane i kontrolowane przez jądro.

Rozdział 1: Podstawowe pojęcia i terminologia

waniem I/O. W istocie jądro jest tą częścią Uniksa, od której zależą właściwości systemu operacyjnego. Wszystkie procesy użytkowników i każdy dostęp do systemu plików powinny być udostępniane, monitorowane i kontrolowane przez jądro.

Funkcje systemowe są wykonywane w ten sam sposób, jak wywołania zwykłych podprogramów czy funkcji w C. Na przykład dane mogą być czytane z pliku za pomocą procedury bibliotecznej C `fread`:

```
nread = fread(inputbuf, OBJSIZE, numberobjs, fileptr);
```

lub za pomocą funkcji systemowej niskiego poziomu `read`:

```
nread = read(filedes, inputbuf, BUFSIZE);
```

Istotna różnica między podprogramem a funkcją systemową polega na tym, że gdy program wywołuje podprogram, jego kod jest zawsze częścią końcowego programu, nawet jeśli został dołączony z biblioteki; przy funkcjach systemowych główna część wykonywanego kodu jest częścią samego jądra, a nie wywołującego programu. Inaczej mówiąc, program wywołujący robi bezpośredni użytku ze środków udostępnianych przez jądro. Przelatczanie pomiędzy procesem użytkownika i jądrem zwykle dokonywane jest za pomocą mechanizmu przerwań programowych.

Nie powinieneś być zaskoczony, dowiadując się, że większość funkcji systemowych wykonuje działania na plikach lub procesach. W rzeczywistości funkcje systemowe tworzą podstawowe pierwotne operacje skojarzone z obydwioma rodzajami obiektów.

W przypadku pliku operacje te mogą zawierać transfer danych do i z pliku, bezpośrednie przeszukiwanie pliku lub zmianę uprawnień dostępu związanych z plikiem.

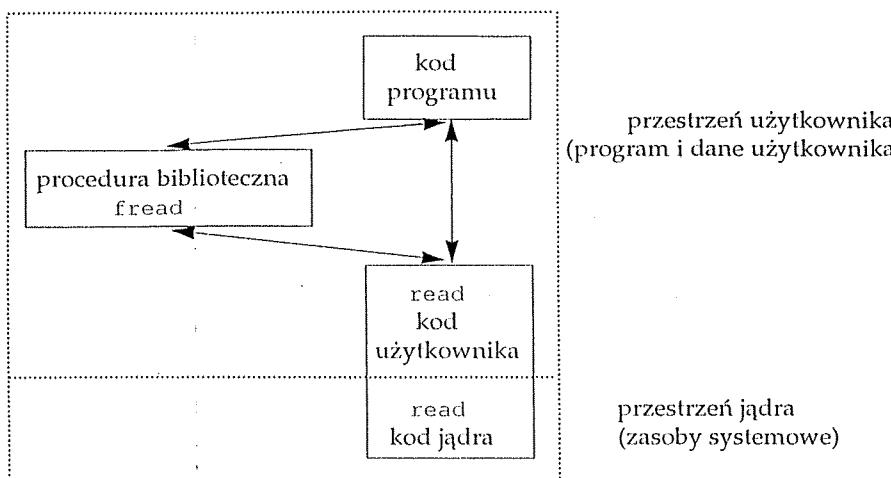
W przypadku procesów operacje funkcji systemowych mogą utworzyć nowy proces, zakończyć istniejący, uzyskać informację o stanie procesu albo ustanowić kanał komunikacyjny pomiędzy dwoma procesami.

Niewielka liczba funkcji systemowych nie robi nic z plikami lub procesami. Najczęściej funkcje systemowe tej kategorii są związane z informacją o systemie lub ze sterowaniem. Na przykład jedna z funkcji systemowych pozwala programowi zapytać jądro o aktualną datę i czas; inna pozwala programowi ponownie ustawić te wartości.

Oprócz interfejsu funkcji systemowych system Unix dostarcza także obszerną bibliotekę standardowych podprogramów. Jednym z najważniejszych przykładów jest **Standardowa Biblioteka I/O** (ang. *Standard I/O Library*). Podprogramy zawarte w tej bibliotece dostarczają środków nie umówionych bezpośrednio przez funkcje systemowe dostępu do pliku, włącznie z konwersjami formatu i automatycznym buforowaniem. Chociaż podprogramy Standardowej Biblioteki I/O gwarantują wydajność, same ostatecznie używają interfejsu funkcji systemowych. Powinny być one postrzegane jako dodatkowa warstwa środków dostępu do pliku, oparta na podstawowych funkcjach systemowych, a nie jako oddzielny podsystem.

W rzeczywistości każdy proces, który nawet w najmniejszy sposób oddziałuje wzajemnie ze swoim środowiskiem, musi używać w pewien sposób funkcji systemowych.

Rysunek 1.2 pokazuje zależność pomiędzy kodem programu a procedurą biblioteczną oraz zależność pomiędzy procedurą biblioteczną a funkcją systemową. Uwidacznia on, że procedura biblioteczna `fread` jest ostatecznie interfejsem do podstawowej funkcji systemowej `read`.



Rysunek 1.2 Zależność pomiędzy kodem programu, procedurą biblioteczną a funkcją systemową

Ćwiczenie 1.1 Wyjaśnij znaczenie następujących terminów:
jdro, funkcja systemowa, procedura w C, proces, katalog, nazwa ścieżki.

ROZDZIAŁ 2

Plik

- 2.1 Elementarne operacje dostępu do pliku
- 2.2 Standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach
- 2.3 Przegląd Standardowej Biblioteki I/O
- 2.4 Zmienna `errno` i funkcje systemowe

2.1 Elementarne operacje dostępu do pliku

2.1.1 Wprowadzenie

W rozdziale przyjrzymy się elementarnym operacjom dostarczonym przez Uniks w celu obsługi plików z wnętrza programów. Operacje te zawierają mały zestaw funkcji systemowych, dających bezpośredni dostęp do dostarczanych przez jdro Uniksa urządzeń I/O. Stanowią one budulec dla wszystkich operacji I/O Uniksa, a każdy inny mechanizm dostępu do pliku jest na nich oparty. Nazwy tych operacji są wymienione w tabeli 2.1. Powtarzanie się czynności wykonywanych przez różne funkcje spowodowane jest rozwojem Uniksa w ciągu ostatniej dekady.

Tabela 2.1 Operacje elementarne Uniksa

nazwa	znaczenie
open	otwiera plik do odczytu lub zapisu, albo tworzy pusty plik
creat	tworzy pusty plik
close	zamyka uprzednio otwarty plik
read	pobiera informacje z pliku
write	umieszcza informacje w pliku
lseek	przechodzi do określonego bajtu w pliku
unlink	usuwa plik
remove	alternatywna metoda usunięcia pliku
fcntl	steruje związanymi z plikiem atrybutami

Typowy program uniksowy powinien wywołać funkcję `open` (lub `creat`) w celu inicjacji pliku, a następnie używać funkcji `read`, `write` i `seek` do manipulacji danymi w tym pliku. Jeśli plik nie jest już potrzebny programowi, powinien on wywołać funkcję `close` wskazując, że zakończył pracę z plikiem. Na koniec, jeśli plik nie jest już potrzebny użytkownikowi, może być całkiem usunięty z systemu operacyjnego przez wywołanie funkcji `unlink` lub `remove`.

Zamieszczony poniżej trywialny program, który po prostu czyta pierwszą część pliku, wyjaśnia tę ogólną strukturę. Ponieważ jest to tylko wstępny przykład, pominieliśmy niektóre, istotne zwykle działania, takie jak obsługa błędów. Ale ostrzegamy, że jest to zła praktyka w odniesieniu do rzeczywistych programów.

```
/* elementarny program przykładowy */

/* te pliki nagłówkowe są omówione poniżej */
#include <fcntl.h>
#include <unistd.h>

main ()
{
    int fd;
    ssize_t nread;
    char buf[1024];

    /* otwiera plik "data" do czytania */
    fd = open("data", O_RDONLY);

    /* wczytuje dane */
    nread = read(fd, buf, 1024);

    /* zamkna plik */
    close (fd);
}
```

Pierwsza instrukcja w przykładzie używa jednej z elementarnych operacji funkcji systemowej:

```
fd = open ("data", O_RDONLY);
```

Powoduje ona otwarcie pliku `data`, znajdującego się w bieżącym katalogu, do użytku programu. Drugi argument w wywołaniu, `O_RDONLY`, jest stałą całkowitą zdefiniowaną w pliku nagłówkowym `<fcntl.h>`, która mówi, żeby system otworzył plik tylko do odczytu (ang. *read only*). Innymi słowy, program będzie mógł tylko czytać zawartość pliku, a nie będzie mógł zniszczyć pliku przez zapis do niego.

Wartość zwracana z wywołania funkcji `open`, umieszczana w zmiennej całkowitej `fd`, jest wyjątkowo ważna. Jeśli wywołanie `open` zakończyło się pomyślnie, `fd` zawiera coś, co jest nazywane **deskryptorem pliku** (ang. *file descriptor*). Deskryptor pliku to nieujemna liczba całkowita, której wartość jest określana przez system. Identyfikuje ona otwarty plik i jest przekazywana jako parametr do innych elementarnych operacji dostępu do pliku, takich jak `read`, `write`, `lseek` i `close`. Jeśli wywołanie `open` zakończy się niepowodzeniem, zwróci wartość `-1`. Ta liczba zwracana jest przez niemal wszystkie funkcje systemowe w celu wskazania błędu.

Rozdział 2: Plik

W rzeczywistym programie powinniśmy sprawdzać tę wartość i podjąć odpowiednie działanie, jeśli wystąpił błąd.

Gdy plik już jest otwarty, nasz przykładowy program używa funkcji systemowej `read`:

```
nread = read(fd, buf, 1024);
```

Co oznacza: jeśli to możliwe, pobierz 1024 znaki z pliku identyfikowanego przez `fd` i umieść je w tablicy znaków `buf`. Zwracana wartość `nread` podaje liczbę faktycznie przeczytanych znaków, która zwykle będzie wynosiła 1024 albo mniej, jeśli plik jest krótszy niż 1024 bajty. Jeśli coś jest źle, `read` (podobnie jak `open`) powinno zwrócić `-1`.

Zmienna `nread` jest typu `ssize_t`, jak zdefiniowano w `<sys/types.h>`. Jeśli dzis wiesz się, że ten plik nagłówkowy nie został włączony w przykładzie, to wiedz, że nie musielibyśmy go włączać, ponieważ podstawowe typy – takie jak `ssize_t` – są też zdefiniowane w `<unistd.h>`. Typ `ssize_t` to nasz pierwszy przykład różnych specjalnych typów zdefiniowanych dla bezpiecznego korzystania z funkcji systemowych. Jest on zwykle redukowany do podstawowego typu całkowitego (i naprawdę w pierwszym wydaniu tej książki¹ `nread` była typu `int` – no, ale tamten niegdyś świat był nieco prostszy).

Ta instrukcja demonstruje inny ważny punkt: elementarne operacje dostępu do pliku mają do czynienia z prostymi, liniowymi sekwencjami znaków lub bajtów. Wywołanie funkcji `read` nie wykonuje żadnej użytecznej konwersji, jak na przykład zamiana znakowej reprezentacji liczby całkowitej na postać używaną wewnętrznie przez komputer. Operacje elementarne `read` i `write` nie powinny być mylone z identycznie nazwanymi procedurami występującymi w językach wysokiego poziomu jak Fortran czy Pascal. Funkcja `read` jest typowa dla filozofii podstaw interfejsu funkcji systemowych: wykonuje pojedynczą, prostą funkcję i dostarcza budulca do tworzenia innych procedur.

Na zakończenie przykładu plik jest zamknięty :

```
close (fd);
```

Mówi to systemowi, że program zakończył współpracę z plikiem skojarzonym z `fd`. Łatwo jest zauważyc, że wywołanie funkcji `close` stanowi odwrotność wywołania `open`. Właściwie, ponieważ program tak czy owak akurat ma się zakończyć, wywołanie `close` nie jest niezbędne, gdyż wszystkie otwarte pliki są automatycznie zamknięte przy zakończeniu procesu. Jednak używanie `close` jest dobrym nawykem.

Ten krótki przykład stanowi przedsmak elementarnych operacji dostępu do pliku. Teraz omówimy każdą z nich bardziej szczegółowo.

2.1.2 Funkcja systemowa `open`

Zanim istniejący plik może być czytany lub zapisywany, musi być otwarty za pomocą funkcji systemowej `open`. Poniższy tekst pokazuje, jak jest ona używana. W celu klarowności i spójności z dokumentacją systemu, wszystkie fragmenty dotyc-

¹ Por. przyp. ze s. VII.

częce zastosowania używają struktury prototypów funkcji ANSI C. Specyfikują one pliki nagłówkowe, w których znajduje się właściwy prototyp i w których są zdefiniowane użyteczne stałe.

Użycie

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

int open(const char *pathname, int flags, [mode_t mode]);
```

Pierwszy argument, pathname, jest wskaźnikiem do napisu zawierającego nazwę ścieżki pliku, który ma być otwarty. Wartość wskazywana przez pathname może być bezwzględną nazwą ścieżki:

/usr/keith/junk

podającą położenie pliku w stosunku do katalogu głównego. Może to być też względna nazwa ścieżki określająca położenie pliku w stosunku do bieżącego katalogu roboczego, na przykład:

keith/junk

lub po prostu:

junk

Oczywiście w ostatnim przypadku program powinien otworzyć plik junk w bieżącym katalogu roboczym. Zasadniczo, ilekroć funkcja systemowa lub podprogram biblioteczny pobiera w argumencie nazwę pliku, akceptuje każdą prawidłową nazwę ścieżki Uniksa.

Drugi argument open, nazywany w naszym opisie użycia flags, jest typu całkowitego i określa metodę dostępu. Wartość flags jest pobierana ze stałej zdefiniowanej w systemowym pliku nagłówkowym <fcntl.h> za pomocą dyrektywy preprocesora #define (przy okazji – fcntl oznacza file control). Jak większość standardowych plików nagłówkowych, <fcntl.h> zwykle znajduje się w katalogu /usr/include i może być włączony do programu za pomocą dyrektywy:

```
#include <fcntl.h>
```

W pliku <fcntl.h> są zdefiniowane trzy stałe, które nas bezpośrednio interesują:

O_RDONLY otwórz plik tylko do odczytu

O_WRONLY otwórz plik tylko do zapisu

O_RDWR otwórz plik do odczytu i zapisu

Jeśli wywołanie funkcji open zakończy się pomyślnie i plik zostanie otwarty, wartość zwracana przez open powinna zawierać nieujemną liczbę całkowitą – deskryptor pliku. Wartość deskryptora pliku jest najmniejszą nieujemną liczbą całkowitą, nie używaną jeszcze jako deskryptor pliku przez proces dokonujący wywołania. Nie musisz znać tej wartości. Jak widzieliśmy na początku rozdziału, jeśli wystąpi błąd, funkcja open powinna zwrócić wartość -1. Może się tak zdarzyć, jeśli na przykład plik nie istnieje. Aby utworzyć nowy plik, programista powinien

użyć wywołania funkcji open z parametrem flags ustawionym na O_CREAT, co opisujemy w następnej części.

Opcjonalny trzeci parametr mode jest znacznikiem używanym tylko z O_CREAT i ponownie będzie omawiany w następnej części – jest on związany z uprawnieniami bezpieczeństwa pliku. Zauważ, że użycie nawiasów kwadratowych w naszym opisie użycia wskazuje na *opcjonalny* charakter mode. Nie potrzebujesz go zawsze stosować.

Poniższy zarys programu otwiera plik junk do czytania i zapisu oraz sprawdza, czy podczas otwarcia nie wystąpił błąd. To bardzo ważne; dobrą zasadą jest wbudowywanie sprawdzania błędów do wszystkich programów, używających funkcji systemowych, ponieważ jeśli nawet coś pójdzie nie tak, aplikacja będzie działać prawidłowo. W przykładzie korzystamy z procedur bibliotecznych printf do wyświetlenia komunikatów i exit do zakończenia procesu wywołującego. Obie procedury są dostarczane standardowo z każdym systemem Unix.

```
#include <stdlib.h>                                /* dla wywołania exit */
#include <fcntl.h>

char *workfile="junk";                               /* definiuje nazwę pliku roboczego */

main()
{
    int filedes;

    /* otwórz stosując O_RDWR z <fcntl.h>      */
    /* plik jest otwierany dla odczytu/zapisu */

    if((filedes = open(workfile, O_RDWR)) == -1)
    {
        printf("Couldn't open %s\n", workfile);
        exit(1);                                /* błąd, więc wyjście */
    }

    /* tu dalszy ciąg programu */

    exit (0);                                     /* normalne wyjście */
}
```

Zauważ, że używamy exit z argumentem 1, kiedy wystąpił błąd, i z argumentem 0 przy pomyślnym zakończeniu. Jest to zgodne z konwencją Uniksa i należy do dobrej praktyki programowania. Jak zobaczymy w dalszych rozdziałach, argument przekazywany do exit (stan wyjścia (ang. exit status) programu) może być dostępny po zakończeniu wykonania programu. Zwróć także uwagę na użycie pliku nagłówkowego <stdlib.h>, zawierającego prototyp wywołania funkcji exit.

Zastrzeżenia

Do naszych rozważań możemy dodać kilka zastrzeżeń. Po pierwsze – zauważ, że liczba plików, które mogą być jednocześnie otwarte przez wykonywany program, jest ograniczona. POSIX (i stąd XSI) określa minimum na 20. Gdy mamy otwartą zbliżoną liczbę plików, możemy użyć funkcji systemowej close do zasygnali-

zowania systemow i, ze skończyliśmy współpracę z plikiem. Funkcję tę omawiamy w podrozdziale 2.1.5. Istnieje też systemowe ograniczenie sumarycznej liczby plików, które mogą być otwarte przez wszystkie procesy razem, określane przez wielkość tabeli wewnętrz jądra.

Po drugie – ku przestrodze: we wczesnych wersjach Uniksa plik nagłówkowy `<fcntl.h>` nie istniał i w parametrze `flags` były używane wartości numeryczne. Istnieje jeszcze dość powszechna, choć nie całkiem zadowalająca, praktyka używania tych wartości numerycznych zamiast nazw stałych zdefiniowanych w `<fcntl.h>`. Możesz więc zobaczyć taką instrukcję, jak:

```
filedes = open(filename, 0);
```

która w normalnych warunkach otwiera plik z dostępem tylko do czytania i jest równoważna z:

```
filedes = open(filename, O_RDONLY);
```

Ćwiczenie 2.1 Utwórz opisany powyżej mały zarys programu. Przetestuj go, gdy plik `junk` nie istnieje. Następnie utwórz `junk` za pomocą swojego ulubionego edytora i ponownie uruchom program. Zawartość pliku `junk` jest całkowicie dowolna.

2.1.3 Tworzenie pliku za pomocą open

Wywołanie funkcji `open` może być także wykorzystane do utworzenia nowego pliku, jak poniżej:

```
filedes = open("/tmp/newfile", O_WRONLY | O_CREAT, 0644);
```

Nowy znacznik `O_CREAT` jest tu połączony z `O_WRONLY`, w celu poinstruowania funkcji `open`, aby utworzyła plik `/tmp/newfile`. Jeśli plik `/tmp/newfile` jeszcze nie istnieje, powinien być utworzony jako plik o zerowej długości i otwarty tylko do zapisu.

Ten przykład wprowadza trzeci parametr wywołania funkcji `open`, który stanowi `mode`. Jest on potrzebny tylko do utworzenia pliku. Nie wchodząc w szczegóły, powiemy tylko, że `mode` zawiera liczbę, określającą **prawa dostępu** (ang. *access permissions*) do pliku. Określają one, którzy użytkownicy systemu mogą odczytywać, zapisywać lub uruchamiać plik. Powyższy przykład używa ósemkowej wartości 0644. Pozwoli to użytkownikowi, który utworzył plik – odczytywać i zapisywać do niego. Inni użytkownicy będą mieli tylko pozwolenie czytania. Sposób tworzenia tej wartości omawiamy w następnym rozdziale. Dla uproszczenia będziemy jej używać bez zmiany w przykładach do końca bieżącego rozdziału.

Następujący zarys programu tworzy plik `newfile` w bieżącym katalogu roboczym:

```
#include <stdlib.h>
#include <fcntl.h>

#define PERMS 0644      /* uprawnienie dla otwarcia z O_CREAT */
char *filename="newfile";

main()
```

```
{
    int filedes;

    if((filedes = open(filename, O_RDWR|O_CREAT, PERMS)) == -1)
    {
        printf ("Couldn't create %s\n", filename);
        exit(1);           /* błąd, więc wyjście */
    }

    /* tu dalszy ciąg programu */

    exit(0);
}
```

Co się stanie, jeśli `newfile` już istnieje? Jeżeli jego prawa dostępu na to pozwalają, będzie otwarty do zapisu, jakby `O_CREAT` nie było wyszczególniane. W tym przypadku parametr `mode` nie odniesie żadnego skutku. Ewentualne połączenie znacznika `O_CREAT` ze znacznikiem `O_EXCL` (wyłącznie) spowoduje niepowodzenie wywołania funkcji `open`, jeśli plik już istnieje. Na przykład:

```
fd = open("lock", O_WRONLY|O_CREAT|O_EXCL, 0644);
```

oznacza: jeśli plik `lock` nie istnieje, wtedy utwórz go z prawami 0644. Jeśli istnieje, zwróć -1 w `fd`, sygnalizując w ten sposób błąd.

Innym użytecznym tu znacznikiem jest `O_TRUNC`. Kiedy używamy go razem z `O_CREAT`, jeśli plik już istnieje i prawa dostępu pozwolą – wymusi obcięcie pliku do zerowej długości. Na przykład:

```
fd = open("file", O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

Jest on niezbędny, jeśli nie chcesz, by dane wyjściowe programu były poprzedzone danymi z poprzednich uruchomień tego programu.

Ćwiczenie 2.2 Ciekawe, że znacznik `O_TRUNC` może być używany samodzielnie, bez `O_CREAT`. Postaraj się przewidzieć wynik, a następnie wypróbuj za pomocą przykładowego programu z użyciem tego znacznika przypadki, gdy plik docelowy istnieje i gdy nie istnieje.

2.1.4 Funkcja systemowa creat

Jako alternatywny sposób tworzenia pliku może być używana funkcja systemowa `creat`. W rzeczywistości jest to funkcja *pierwotna*, ale obecnie nieco nadmiarowa i oferuje mniejszą kontrolę niż `open`. Włączamy ją tu do kompletu. Podobnie jak `open`, zwraca ona albo nieujemny deskryptor pliku, albo -1 przy błędzie. Jeśli zwracana wartość to ważny deskryptor pliku, wtedy plik powinien być otwarty do zapisu. Funkcja `creat` jest wywoływana następująco:

Użycie

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

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

Pierwszy parametr, `pathname`, wskazuje na nazwę ścieżki Uniksa. Podaje on wymaganą nazwę i położenie nowego pliku. Parametr `name` – podobnie jak przy `open` – podaje wymagane prawa dostępu. I podobnie jak przy `open`, jeśli plik już istnieje – wtedy drugi argument jest ignorowany. Jednak w przeciwieństwie do `open`, `creat` zawsze skraca do zera długość istniejącego pliku przed zwrotem jego deskryptora. Oto przykład użycia funkcji `creat`:

```
filedes = creat("/tmp/newfile", 0644);
```

co jest równoważne z:

```
filedes = open("/tmp/newfile", O_WRONLY|O_CREAT|O_TRUNC, 0644);
```

Trzeba tu zwrócić uwagę, że funkcja `creat` zawsze otwiera plik tylko do zapisu. Program nie może na przykład utworzyć pliku za pomocą `creat`, zapisać do niego danych, a następnie cośnąć się i próbować czytać z niego, dopóki nie zamknie pliku i nie otworzy ponownie za pomocą `open`.

Ćwiczenie 2.3 Napisz krótki program, który najpierw tworzy plik używając `creat`, a następnie, bez wywoływanego `close`, natychmiast otwiera go za pomocą funkcji systemowej `open` do odczytu i zapisu. W obu przypadkach program powinien wskazywać sukces albo niepowodzenie operacji, wyświetlając za pomocą `printf` odpowiednie komunikaty.

2.1.5 Funkcja systemowa close

Funkcja systemowa `close` jest odwrotnością `open`. Mówiąc o systemie, że wywołujący proces zakończył pracę z plikiem. Funkcja `close` jest użyteczna z powodu ograniczonej liczby plików, które działający program może jednocześnie trzymać otwarte.

Użycie

```
#include <unistd.h>
int close(int filedes);
```

Funkcja systemowa `close` pobiera tylko jeden argument – deskryptor zamykanego pliku. Będzie on zwykle pochodził z poprzedniego wywołania funkcji `open` albo `creat`. Następujący fragment programu ilustruje prostą zależność pomiędzy `open` i `close`:

```
filedes = open ("file", O_RDONLY);
.
.
.
close(filedes);
```

Funkcja systemowa `close` zwraca 0, jeśli wywołanie zakończyło się pomyślnie i -1 przy błędzie (co może się zdarzyć, jeśli argument całkowity nie jest ważnym deskryptorem pliku).

Zauważ, że aby zapobiec całkowitemu chaosowi, wszystkie otwarte pliki są automatycznie zamknięte, gdy program kończy się wykonywać.

2.1.6 Funkcja systemowa read

Funkcji systemowej `read` używamy do kopiowania pod kontrolą wywołującego programu określonej liczby znaków albo bajtów z pliku do bufora. Bufor jest formalnie deklarowany jako wskaźnik do `void`, co oznacza, że może zawierać elementy dowolnego typu. Chociaż zwykle bufor jest tablicą `char`, również dobrze może też być tablicą struktur zdefiniowanych przez użytkownika `structs`.

Zauważ, że programiści C mają skłonność do wymiennego używania terminów „znaki” i „bajty”. Bajt jest jednostką pamięci potrzebną do zapamiętania znaku i w większości komputerów ma długość ośmiu bitów. Termin „znak” zwykle opisuje element zestawu znaków ASCII, składający się z siedmiobitowego wzorca. Dlatego bajt może najczęściej pamiętać więcej wartości niż jest znaków ASCII – w sytuacji, gdy masz do czynienia z danymi dwójkowymi. Typ `char` w języku C reprezentuje bardziej ogólnie pojęcie bajtu, a jego nazwa jest raczej niewłaściwa.

Użycie

```
#include <unistd.h>
ssize_t read(int filedes, void *buffer, size_t n);
```

Pierwszy parametr, `filedes`, jest deskryptorem pliku, otrzymanym z uprzedniego wywołania funkcji `open` albo `creat`. Drugi parametr, `buffer`, to wskaźnik do tablicy lub struktury, do której będą kopowane dane. W wielu przypadkach będzie on po prostu nazwą samej tablicy. Na przykład:

```
int fd;
ssize_t nread;
char buffer[SOMEVALUE];

/* fd uzyskany z wywołania funkcji open */
.

.

nread = read(fd, buffer, SOMEVALUE);
```

Jak możesz stwierdzić na tym przykładzie, trzeci parametr `read` jest liczbą dodatnią (zdefiniowaną jako specjalny typ `size_t`), określającą liczbę bajtów, które mają być odczytane z pliku.

Liczba zwracana przez `read` (powyżej przypisywana do `nread`) rejestruje liczbę rzeczywiście przeczytanych bajtów. Zwykle będzie to liczba znaków wymagana przez program, ale nie jest tak zawsze i `nread` może przybierać mniejsze wartości. W dodatku, kiedy wystąpi błąd, `read` powinno zwrócić wartość -1. Zdarza się to na przykład, kiedy do funkcji `read` został przekazany nieważny deskryptor pliku.

Wskaźnik odczytu-zapisu

Program może w dół naturalny sposób, wywołując kolejno funkcję `read`, przejrzeć sekwencyjnie plik. Na przykład jeśli założymy, że plik `foo` zawiera co najmniej 1024 znaki, następujący fragment powinien umieścić pierwsze 512 znaków z `foo` w `buf1`, a następnie 512 znaków w `buf2`:

```
int fd;
ssize_t n1, n2;
char buf1[512], buf2[512];

if(( fd = open("foo", O_RDONLY)) == -1)
    return (-1);

n1 = read(fd, buf1, 512);
n2 = read(fd, buf2, 512);
```

System śledzi pozycję procesu w pliku za pomocą jednostki nazywanej **wskaźnikiem odczytu-zapisu** (ang. *read-write pointer*), a czasem po prostu **wskaźnikiem pliku** (ang. *file pointer*). Zasadniczo rejestruje on położenie następnego bajtu w pliku, który powinien być odczytany (lub zapisany), za pomocą określonego deskryptora pliku, który przypomina nieco zakładkę w książce. Jego wartość jest pamiętana wewnętrznie przez system i programista nie musi jawnie przypisywać zmiennej do jego zapamiętania. Bezpośredni dostęp do pliku, przy którym pozycja wskaźnika odczytu-zapisu jest jawnie zmieniana, może być uzyskany za pomocą funkcji systemowej `lseek`, opisanej w części 2.1.10. W przypadku `read` system po prostu powiększa wskaźnik odczytu-zapisu o liczbę bajtów przeczytanych w każdym wywołaniu funkcji.

Ponieważ funkcja `read` może być używana do przeglądania pliku od początku do końca, program musi mieć możliwość wykrycia końca pliku. I dlatego ważna jest wartość zwieracana przez `read`. Kiedy liczba znaków żądanych w wywołaniu funkcji `read` jest większa niż liczba znaków pozostałych w pliku, system przekaże tylko pozostałe znaki, odpowiednio ustawiając zwieracaną wartość. Dalsze wywołania funkcji `read` powinny zwieracać wartość 0. Dzieje się tak dlatego, że nie zostały żadne znaki do odczytu. I rzeczywiście sprawdzanie, czy `read` nie zwierac w wartości 0, jest zwykłym sposobem testowania końca pliku w programie lub co najmniej w programie, który używa `read`.

Następny przykładowy program, `count`, łączy omówione powyżej zagadnienia:

```
/* count -- liczy znaki w pliku */

#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define BUFSIZE 512

main()
{
    char buffer[BUFSIZE];
```

```
int filedes;
ssize_t nread;
long total = 0;

/* otwórz "anotherfile" tylko do czytania */
if(( filedes = open("anotherfile", O_RDONLY)) == -1)
{
    printf("error in opening anotherfile\n");
    exit(1);
}

/* pętla aż do EOF, pokazanego przez zwrot wartości 0 */
while(( nread = read(filedes, buffer, BUFSIZE)) >0)
    total += nread; /* powiększ total */

printf ("total chars in anotherfile: %ld\n", total);
exit(0);
```

Ten program będzie czytał plik `anotherfile` fragmentami po 512 znaków. Po każdym wywołaniu funkcji `read` program powiększa zmienną `total` o liczbę znaków faktycznie skopiowanych do tablicy `buffer`. (Jak sądzisz, dlaczego `total` jest zadeklarowana jako `long integer`?)

Używamy wartości 512 dla liczby czytanych znaków, ponieważ system Unix pracuje bardziej efektywnie, jeśli przesyła dane w blokach o wielkości będącej wielokrotnością bloku dyskowego – w tym przypadku 512 bajtów. (Rzeczywista wielkość bloku zależy od systemu i może wynosić 8K lub więcej). Jednak możemy podać `read` dowolną liczbę, która przyjdzie nam na myśl, nawet 1. Nie ma żadnej innej korzyści ze stosowania wartości właściwej dla danego systemu poza efektywnością, ale jak zobaczymy w podrozdziale 2.1.9 – poprawa efektywności może być znaczna.

Aby wykorzystać rzeczywistą wielkość bloku dyskowego w swoim systemie, możesz użyć definicji `BUFSIZ` z pliku `/usr/include/stdio.h` (który jest powiązany z dobrze znaną Standardową Biblioteką I/O). Na przykład:

```
#include <stdio.h>
.

.

nread = read(filedes, buffer, BUFSIZ);
```

Ćwiczenie 2.4 Zmodyfikuj program `count`, aby akceptował parametry wiersza poleceń zamiast używania stałej nazwy pliku. Przetestuj go na małym pliku, z kilkoma liniami tekstu.

Ćwiczenie 2.5 Zmodyfikuj program `count`, aby wyświetlał liczbę słów i linii w pliku. Zdefiniuj słowo jako będące albo znakiem przestankowym, albo ciągiem alfanumerycznym nie zawierającym znaków odstępu (białej spacji) jak spacja, tabulacja lub znak nowego wiersza. Linia jest oczywiście sekwencją znaków zakończoną znakiem nowego wiersza.

2.1.7 Funkcja systemowa write

Funkcja systemowa `write` stanowi naturalną odwrotność `read`. Kopiuje ona dane z bufora programu, najczęściej również zadeklarowanego jako tablica, do zewnętrznego pliku.

Użycie

```
#include <unistd.h>
ssize_t write(int filedes, const void *buffer, size_t n);
```

Podobnie jak `read`, funkcja `write` pobiera trzy argumenty: `filedes` – deskryptor pliku; `buffer` – wskaźnik do danych, które mają być zapisane; `n` – dodatnią liczbę określającą liczbę znaków do zapisania. Wartość zwracana jest liczbą znaków faktycznie skierowanych przez `write` na wyjście albo kodem błędu `-1`. W rzeczywistości, jeśli nie jest to `-1`, zwracana wartość powinna być prawie zawsze równa `n`. Jeśli jest ona mniejsza, oznacza to, że coś przebiegło niewłaściwie. Może się tak zdarzyć na przykład, gdy wywołanie funkcji `write` zapelnia nośnik wyjściowy, zanim zakończy swoją pracę. (Jeśli nośnik jest pełny, zanim jeszcze zostało dokonane wywołanie funkcji `write`, wtedy powinno być zwrócone `-1`).

Wywołania funkcji `write` często używa się z deskryptorem pliku, uzyskanym dla nowo otwartego pliku. W tym przypadku łatwo jest zobaczyć, co się stanie. Plik ma początkowo zerową długość (został świeżo utworzony albo obcięty) i każde wywołanie `write` po prostu dodaje dane na końcu pliku, ze wskaźnikiem odczytu-zapisu ustawnionym na następną pozycję po ostatnim zapisanym bajcie. Na przykład fragment:

```
int fd;
ssize_t w1, w2;
char header1[512], header2[1024];

if(( fd = open("newfile", O_WRONLY|O_CREAT|O_EXCL, 0644)) == -1)
    return (-1);

w1 = write(fd, header1, 512);
w2 = write(fd, header2, 1024);
```

powinien utworzyć plik o długości 1536 bajtów, zawierający kolejno zawartość `header1` i `header2`.

Co się stanie, jeśli program otworzy istniejący plik do zapisu i natychmiast zacznie do niego zapis? Odpowiedź jest bardzo prosta: stare dane w pliku będą nadpisane przez nowe, znak po znaku. Na przykład założmy, że plik `oldhat` ma 500 znaków długości. Jeśli program otworzy `oldhat` do zapisu w zwykły sposób, a następnie zapisze 10 znaków, pierwsze 10 znaków `oldhat` zostanie zastąpione przez zawartość bufora zapisu programu. Następne takie wywołanie funkcji `write` zastąpi kolejne 10 znaków, i tak dalej. Gdy zostanie osiągnięty koniec oryginal-

Rozdział 2: Plik

nego pliku, plik – zawierający teraz nowe dane – będzie rozszerzany z każdym kolejnym wywołaniem funkcji `write`. Aby tego uniknąć, plik powinien być otwarty ze znacznikiem `O_APPEND`, na przykład:

```
filedes = open(filename, O_WRONLY|O_APPEND);
```

Jeśli `open` powiedzie się, wskaźnik odczytu-zapisu zostanie umieszczony tuż za ostatnim bajtem pliku i każde wywołanie funkcji `write` powinno dodawać dane na końcu pliku. Jest to wyjaśnione szerzej w podrozdziale 2.1.12.

2.1.8 Przykład copyfile

W ten sposób dotarliśmy do miejsca, zawierającego nasz pierwszy przykład praktyczny. Zadanie polega na napisaniu funkcji `copyfile`, która powinna kopiować zawartość jednego pliku do drugiego. Zwracaną wartością powinno być zero wskazujące sukces lub liczba ujemna wskazująca błąd.

Elementarna logika dyktuje następujące postępowanie: otwórz pierwszy plik, następnie utwórz drugi; czytaj z pierwszego pliku i zapisuj do drugiego aż do osiągnięcia końca pierwszego pliku. Na koniec zamknij oba pliki.

Ukończoną rozwiązanie może wyglądać następująco:

```
/* copyfile – kopiuje name1 do name2 */
#include <unistd.h>
#include <fcntl.h>

#define BUFSIZE      512 /* wielkość czytanego fragmentu */
#define PERM          0644 /* prawa dostępu dla nowego pliku */

/* kopiuje name1 do name2 */
int copyfile( const char *name1, const char *name2)
{
    int infile, outfile;
    ssize_t nread;
    char buffer[BUFSIZE];

    if(( infile = open(name1, O_RDONLY )) == -1)
        return (-1);

    if(( outfile = open(name2, O_WRONLY|O_CREAT|O_TRUNC, PERM)) == -1)
    {
        close(infile);
        return (-2);
    }

    /* teraz czytaj z name1 po BUFSIZE znaków naraz */
    while(( nread = read(infile, buffer, BUFSIZE)) > 0)
    {
        /* zapisz bufor do pliku wyjściowego */
        if( write(outfile, buffer, nread) < nread)
        {
            close(infile);
            return (-3);
        }
    }
}
```

```

        close(outfile);
        return (-3);           /* błąd zapisu */

    }

    close(infile);
    close(outfile);

    if( nread == -1)
        return (-4);          /* błąd w ostatnim odczycie */
    else
        return (0);           /* wszystko w porządku */
}

```

Teraz można użyć `copyfile` za pomocą wywołania:

```
retcode = copyfile("squarepeg", "roundhole");
```

Ćwiczenie 2.6 Zmodyfikuj `copyfile` tak, żeby akceptowała jako parametry raczej dwa deskryptory plików niż dwie nazwy plików. Przetestuj tę nową wersję.

Ćwiczenie 2.7 Jeśli znasz argumenty wiersza poleceń, wykorzystaj jedną z procedur `copyfile` do utworzenia programu `mycp`, który kopiuje jeden argument będący nazwą pliku do drugiego.

2.1.9 Funkcje read, write i efektywność

Procedura `copyfile` dostarcza sposobu mierzenia efektywności elementarnych operacji dostępu do pliku w zależności od wielkości bufora. Jednym ze sposobów jest po prostu skompilowanie `copyfile` z różnymi wartościami `BUFSIZE`, a następnie pomiar czasu wynikowego programu za pomocą polecenia `time` Uniksa. Zrobiliśmy to, używając następującej funkcji `main`:

```
/* funkcja main do testowania "copyfile" */

main()
{
    copyfile("test.in", "test.out");
}
```

Tabela 2.2 Wyniki testu `copyfile`

BUFSIZE	czas rzeczywisty	czas użytkownika	czas systemu
1	0:24.49	0:3.13	0:21.16
64	0:0.46	0:0.12	0:0.33
512	0:0.12	0:0.02	0:0.08
4096	0:0.07	0:0.00	0:0.05
8192	0:0.07	0:0.01	0:0.05

W tabeli 2.2 pokazujemy wyniki otrzymane przy kopiowaniu tego samego dużego pliku (68307 bajtów) na komputerze z Uniksem SVR4 z naturalnym współczynnikiem zablokowania dysku równym 512. Format tabeli odzwierciedla wyjście polecenia `time`. Pierwsza kolumna podaje wartość `BUFSIZE`. Druga kolumna podaje rzeczywisty albo aktualny czas działania procesu w minutach, sekundach i setnych częściach sekundy. Trzecia kolumna podaje czas użytkownika, będący czasem zajętym przez te części programu, które nie są funkcjami systemowymi. Na skutek ziarnistości zegara używanego przez `time`, jedna z pozycji w tej kolumnie jest błędnie relacjonowana jako zerowa. Czwarta i ostatnia kolumna podaje ilość czasu, którą jądro spędzało obsługując funkcje systemowe. Jak możesz zobaczyć, wartości z trzeciej i czwartej kolumny po zsumowaniu nie dają rzeczywistego czasu. Dzieje się tak, ponieważ w systemie Unix działa szereg procesów jednocześnie. W związku z tym system nie spędzał całego czasu na obsłudze twojego programu!

Nasze wyniki są niezbite; odczyt i zapis po jednym bajcie naraz daje bulwersującą wydajność, podczas gdy zwiększenie wielkości bufora znacznie ja powiększa. Najlepsza wydajność jest osiągana, gdy `BUFSIZE` stanowi wielokrotność naturalnego współczynnika zablokowania dysku systemu, jak pokazują wyniki dla wartości `BUFSIZE` 512, 4096 i 8192 bajty.

Powinniśmy też zaakcentować, że większość (ale nie całość) każdego wzrostu efektywności pochodzi po prostu ze zmniejszenia liczby funkcji systemowych. Przelatczanie trybu pomiędzy programem a jądrem w czasie wykonywania funkcji systemowych może być dość kosztowne. Zasadniczo powinieneś jak najbardziej zmniejszyć liczbę wykonywanych przez program funkcji systemowych, jeśli potrzebujesz bardzo dużej wydajności.

2.1.10 Funkcja lseek i bezpośredni dostęp

Funkcja systemowa `lseek` umożliwia użytkownikowi zmianę pozycji wskaźnika odczytu-zapisu, to jest zmianę numeru bajtu, który będzie następnie odczytany lub zapisany. Dlatego `lseek` umożliwia bezpośredni dostęp do pliku.

Użycie

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

off_t lseek(int filedes, off_t offset, int start_flag);
```

Pierwszy parametr, `filedes`, jest deskryptorem otwartego pliku. Drugi parametr, `offset`, rzeczywiście określa nową pozycję wskaźnika odczytu-zapisu. Podaje on liczbę bajtów, które należy dodać do pozycji początkowej. Trzeci argument, wartość całkowita `start_flag`, określa, co jest pozycją początkową. Wyszczególnia ona, odkąd ma być mierzony `offset` w pliku. Argument `start_flag` może przybrać pokazane poniżej wartości symboliczne (z `<unistd.h>`):

SEEK_SET offset jest mierzony od początku pliku; zwykle jest to wartość całkowita = 0

SEEK_CUR offset jest mierzony od bieżącego położenia wskaźnika pliku; zwykłe wartość = 1

SEEK_END offset jest mierzony od końca pliku; zwykłe wartość = 2

Widać to lepiej na rysunku 2.1, który przedstawia plik 7-bajtowy.

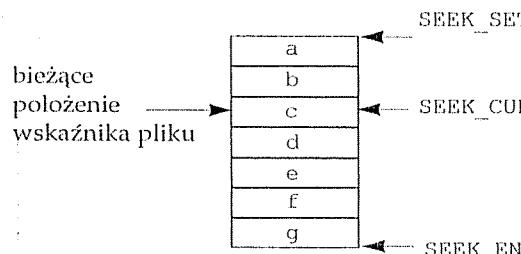
Oto przykład użycia lseek:

```
off_t newpos;
```

```
.
```

```
newpos = lseek(fd, (off_t)-16, SEEK_END);
```

który ustawia pozycję 16 bajtów przed końcem pliku. Zwróć uwagę na rzutowanie -16 bajtów do typu (off_t).



Rysunek 2.1 Symboliczne wartości start_flag

We wszystkich przypadkach zwracana wartość (w tym przykładzie zawarta w newpos) podaje nową pozycję w pliku. Jeśli wystąpi błąd, powinna ona zawierać zwykły kod błędu -1.

Jest tu jeszcze pewna ilość spraw do odnotowania. Po pierwsze, newpos i offset są typu off_t, jak zdefiniowano w <sys/types.h>, który powinien być typem wystarczająco dużym, aby radzić sobie z każdym plikiem w systemie. Po drugie, offset może być ujemny, jak pokazano w przykładzie. Inaczej mówiąc, możliwe jest przesunięcie wstecz od punktu startowego, wskazanego przez start_flag. Błąd powstanie tylko wtedy, gdy spróbujesz przesunąć wskaźnik na pozycję przed początkiem pliku. Po trzecie, możliwe jest wyszczególnienie pozycji poza końcem pliku. Jeśli zostanie to zrobione, nie będzie tam oczywiście żadnych danych czekających na odczyt – Unix nie umożliwia jeszcze podróży w czasie – ale kolejna funkcja write jest doskonale określona i spowoduje rozszerzenie pliku. Każda pusta przestrzeń między starym końcem pliku a początkową pozycją nowych danych może nie być fizycznie przydzielona, ale w kolejnych wywołaniach funkcji read będzie wydawała się wypełniona znakiem zerowym ASCII.

Jako prosty przykład możemy zbudować fragment programu, który powinien dodawać dane na końcu istniejącego pliku przez otwarcie pliku, przesunięcie do końca pliku za pomocą lseek i rozpoczęcie zapisu:

```
filedes = open(filename, O_RDWR);
lseek(filedes, (off_t)0, SEEK_END);
write(filedes, outbuf, OBSIZE);
```

Parametr kierunku dla lseek jest ustawiony na SEEK_END, w celu przemieszczenia wskaźnika odczytu-zapisu do końca pliku. Ponieważ nie chcemy żadnych dalszych przemieszczeń, przesunięcie ma wartość zerową.

W ten sam sposób można wywołać funkcję lseek w celu uzyskania rozmiaru pliku, ponieważ zwraca ona nową pozycję w pliku.

```
off_t filesize;
int filedes;

.
.

filesize = lseek(filedes, (off_t)0, SEEK_END);
```

Ćwiczenie 2.8 Napisz funkcję, która używa lseek do otrzymania rozmiaru otwartego pliku, ale nie zmienia wartości wskaźnika odczytu-zapisu.

2.1.11 Przykład hotelu

Podamy teraz wymyślony, ale wiele wyjaśniający przykład. Założmy mianowicie, że mamy plik residents do rejestrowania nazw stałych mieszkańców hotelu. Linia 1 zawiera nazwisko mieszkańca pokoju 1, linia 2 nazwisko mieszkańca pokoju 2 i tak dalej (jak możesz zauważyć, jest to hotel z wyjątkowo dobrym systemem numeracji pokoi). Każda linia ma dokładnie 41 znaków długości, pierwsze 40 znaków zawiera nazwisko mieszkańca, a 41 – to znak nowego wiersza, umożliwiający wyświetlenie pliku za pomocą polecenia cat. Uniksa.

Następująca funkcja getoccupier powinna, mając podany całkowity numer pokoju, obliczyć położenie pierwszego bajtu nazwiska mieszkańca, następnie przesunąć wskaźnik do tej pozycji i odczytać zawarte dane. Funkcja zwraca wskaźnik do napisu zawierającego nazwisko mieszkańca lub wskaźnik zerowy (null) w przypadku błędu (użyjemy wartości NULL zdefiniowanej w pliku <stdio.h>). Zauważ, że zmiennej deskryptora pliku infile nadajemy początkową wartość -1. Przez testowanie tej wartości możemy zapewnić, że plik jest otwierany tylko jeden raz.

```
/* getoccupier -- weź nazwisko mieszkańca z pliku rezydentnego */

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

#define NAMELENGTH 41

char namebuf[NAMELENGTH];
int infile = -1; /* bufor do przechowywania nazwiska */
                  /* powinien przechować deskryptor pliku */

char *getoccupier(int roomno)
{
    off_t offset;
```

```

ssize_t nread;

/* otwórz plik za pierwszym razem */
if( infile == -1 )
    (infile = open("residents", O_RDONLY) ) == -1)
{
    return (NULL);           /* nie można otworzyć pliku */
}

offset = (roomno - 1) * NAMELENGTH;

/* znajdź zapis pokoju i wczytaj nazwę mieszkańców */
if(lseek(infile, offset, SEEK_SET) == -1)
    return (NULL);

if(( nread = read(infile, namebuf, NAMELENGTH)) <= 0)
    return (NULL);

/* utwórz napis zamieniając znak nowego wiersza końcowym zerem */
namebuf[nread - 1] = '\0';
return (namebuf);
}

Jeżeli założymy, że hotel zawiera 10 pokoi, następujący program powinien kolejno wywoływać getoccupier w celu przeglądania pliku, używając procedury printf ze Standardowej Biblioteki I/O do wyświetlenia każdego znalezionej nazwiska:

/* listoc – wypisuje nazwiska wszystkich mieszkańców */

#define NROOMS      10

main ()
{
    int j;
    char *getoccupier(int), *p;

    for( j = 1; j <= NROOMS; j++)
    {
        if(p = getoccupier(j))
            printf("Room %2d, %s\n", j, p);
        else
            printf("Error on room %d\n", j);
    }
}

```

Ćwiczenie 2.9 Opracuj mechanizm rozstrzygnięcia, czy pokój jest pusty. Zmodyfikuj getoccupier, i jeśli to konieczne – plik danych, aby to uwzględnił. Teraz napisz procedurę o nazwie findfree do znalezienia wolnego pokoju o największym numerze.

Ćwiczenie 2.10 Napisz procedurę freeroom do usuwania gościa z jego pokoju. Następnie napisz addguest do umieszczania nowego gościa w pokoju, sprawdzając wcześniej, czy pokój jest pusty.

Ćwiczenie 2.11 Włącz getoccupier, freeroom, addguest i findfree do prostej programu narzędziowego frontdesk, który utrzymuje plik danych. Użyj argumentów wiersza poleceń albo napisz program interakcyjny, który wywołuje printf i getchar. W obu przypadkach będziesz potrzebował sposobu zamiany ciągu znaków na liczbę całkowitą, aby obliczyć numer pokoju. Możesz użyć procedury bibliotecznej atoi, jak poniżej:

```
i = atoi(string);
```

gdzie string jest wskaźnikiem znakowym, a i. liczbą całkowitą.

Ćwiczenie 2.12 Jako bardziej ogólne ćwiczenie napisz program wykorzystujący lseek, który kopiuje bajty z jednego pliku do innego w odwrotnej kolejności. Czy twoje rozwiązanie jest efektywne?

Ćwiczenie 2.13 Używając lseek, napisz procedurę kopiowania ostatnich 10 znaków, ostatnich 10 słów i ostatnich 10 linii jednego pliku do innego.

2.1.12 Dodawanie danych do pliku

Z podrozdziału 2.1.10 można wywnioskować, że do dodawania danych na końcu pliku służy następujący kod:

```
/* ustaw na koniec pliku */
lseek(filedes, (off_t)0, SEEK_END);
write(filedes, appbuf, BUFSIZE);
```

Jednak bardziej zręczne jest użycie dodatkowego znacznika dla open, o nazwie O_APPEND. Jeśli jest ustawiony, O_APPEND powoduje pozycjonowanie wskaźnika pliku na koniec pliku, ilekroć ma miejsce zapis. Jest to użyteczne, jeśli programista chce po prostu dodać dane na końcu pliku i zabezpieczyć oryginalną zawartość przed przypadkowym zniszczeniem.

Znacznik O_APPEND może być używany następująco:

```
filedes = open("yetanother", O_WRONLY | O_APPEND);
```

Każde kolejne użycie write, takie jak:

```
write(filedes, appbuf, BUFSIZE);
```

będzie dodawać dane na końcu pliku.

Ćwiczenie 2.14 Napisz procedurę fileopen, która pobiera dwa argumenty; pierwszym jest napis zawierający nazwę pliku, drugim napis (ciąg znaków), który może przybierać następujące wartości:

r otwórz plik tylko do odczytu

w otwórz plik tylko do zapisu

rw otwórz plik do odczytu i zapisu

a otwórz plik do dodawania danych na końcu

Procedura fileopen powinna zwracać deskryptor pliku lub kod błędu -1.

2.1.13 Kasowanie pliku

Istnieją dwie metody eliminacji pliku z systemu. Są nimi funkcje `unlink` i `remove`.

Użycie

```
#include <unistd.h>
int unlink(const char *pathname);
#include <stdio.h>
int remove(const char *pathname);
```

Obie funkcje pobierają jeden argument: napis (ciąg znaków) zawierający nazwę usuwanego pliku, na przykład:

```
unlink("/tmp/usedfile");
remove("/tmp/tmpfile");
```

Obie funkcje zwracają 0 wskazując sukces, albo -1 wskazując niepowodzenie.

Dlaczego dwie funkcje? Początkowo była tylko funkcja `unlink`. Funkcja `remove`, określona w standardzie ANSI C, jest ostatnim dodatkiem do *XSI*. Do usuwania zwykłych plików funkcja `remove` jest identyczna z `unlink`. Do usuwania pustych katalogów funkcja `remove(path)` jest równoważna z `rmdir(path)` – zamiast `unlink` do katalogów powinny być używane inne funkcje systemowe. Z tym tematem spotkamy się jeszcze w rozdziale 4.

2.1.14 Funkcja systemowa `fcntl`

Funkcja systemowa `fcntl` była wprowadzona w celu dostarczenia kontroli nad otwartym już plikiem. Jest to dość dziwne zwierzę, które wykonuje rozmaite funkcje zamiast pojedynczej, dobrze zdefiniowanej roli.

Użycie

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
/* Uwaga: typ ostatniego parametru może się zmieniać,
   jak wskazuje wielokropki "...". */
int fcntl(int filedes, int cmd, ...);
```

Funkcja systemowa `fcntl` działa na otwartym pliku, identyfikowanym przez deskryptor pliku `filedes`. Programista określa konkretną funkcję przez wybór wartości dla parametru całkowitego `cmd` z pliku nagłówkowego `<fcntl.h>`. Rodzaj trzeciego parametru zależy od wartości parametru `cmd`. Na przykład, jeśli funkcja `fcntl` jest używana do ustawienia znacznika stanu pliku, wtedy trzeci parametr będzie liczbą całkowitą. Później przekonamy się, że w przypadku używania funkcji

`fcntl` do zablokowania pliku, trzeci parametr jest wskaźnikiem do `struct flock`. W niektórych przypadkach trzeciego parametru w ogóle się nie używa.

Niektóre z tych funkcji dotyczą interakcji plików oraz procesów i nie będziemy ich tu omawiać; istnieją jednak dwie funkcje, które będą przydatne. Są one identyfikowane przez wartości `cmd` równe `F_GETFL` i `F_SETFL`.

`F_GETFL` poleca `fcntl` zwrócenie aktualnego znacznika stanu pliku, ustawionego przez `open`. Następująca funkcja `filestatus` używa `fcntl` w ten sposób, aby wyświetlić bieżący stan otwartego pliku.

```
/* filestatus -- opisuje bieżący stan pliku */

#include <fcntl.h>

int filestatus(int filedes)
{
    int arg1;

    if(( arg1 = fcntl(filedes, F_GETFL)) == -1)
    {
        printf("filestatus failed\n");
        return (-1);
    }

    printf("File descriptor %d: ", filedes);

    /* porównaj argument ze znacznikami otwarcia */
    switch( arg1 & O_ACCMODE)
    {
        case O_WRONLY:
            printf ("write-only");
            break;
        case O_RDWR:
            printf ("read-write");
            break;
        case O_RDONLY:
            printf ("read-only");
            break;
        default:
            printf ("No such mode");
    }

    if(arg1 & O_APPEND)
        printf("- append flag set");

    printf("\n");
    return (0);
}
```

Zauważ, że testujemy, czy w znaczniku stanu pliku zawartego w `arg1` jest ustawiony konkretny bit, za pomocą bitowego operatora AND (koniekcji bitowej), oznaczonego przez pojedynczy symbol `&`. Bit jest testowany z `O_ACCMODE`, maską zdefiniowaną specjalnie w tym celu w `<fcntl.h>`.

`F_SETFL` jest używany do zmiany powiązanych z plikiem znaczników stanu pliku. Nowe znaczniki są podawane w trzecim argumencie `fcntl`. W ten sposób mogą być ustawiane tylko pewne znaczniki; nie możesz na przykład nagle przestawić pliku otwartego tylko do odczytu na plik otwarty do odczytu i zapisu (dlaczego?). Jednak za pomocą następującego wywołania możesz zapewnić, że wszystkie kolejne zapisy będą dodawane na końcu pliku:

```
if( fcntl(filedes, F_SETFL, O_APPEND) == -1)
    printf("fcntl error \n");
```

2.2 Standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach

2.2.1 Podstawowe pojęcia

Dla każdego wykonywanego programu system Unix automatycznie otwiera trzy pliki. Są to: **standardowe wejście** (ang. *standard input*), **standardowe wyjście** (ang. *standard output*) i **standardowe wyjście komunikatów o błędach** (ang. *standard error*). Zawsze są one identyfikowane odpowiednio przez deskryptory pliku 0, 1 i 2. Pomimo podobnie brzmiących nazw, nie myl ich ze Standardową Biblioteką I/O.

Domyślnie odczyt ze standardowego wejścia powoduje, że program czyta dane z klawiatury. Podobnie zapis do standardowego wyjścia lub standardowego wyjścia komunikatów o błędach domyślnie spowoduje wyświetlenie komunikatu na ekranie terminala. To pierwszy sposób, w jaki elementarne operacje dostępu do pliku mogą być użyte do wszystkich typów operacji I/O, a nie tylko operacji dotyczących plików dyskowych.

Program, który używa tych standardowych deskryptorów pliku, nie jest jednak w żaden sposób przywiązany do terminala. Każdy z deskryptorów może być oddzielnie zmieniony przy wywołaniu programu za pomocą własności przekierowania, dostarczanej przez standardową powłokę Uniksa (procesor poleceń Uniksa). Na przykład polecenie:

```
$ prog_name < infile
```

spowoduje, że program będzie akceptował dane z `infile`, gdy będzie czytał z deskryptora pliku 0, a nie z terminala, będącego normalnym źródłem dla standardowego wejścia.

Wszystkie dane zapisywane do standardowego wyjścia mogą być w podobny sposób przekierowane do pliku wyjściowego. Na przykład:

```
$ prog_name > outfile
```

Prawdopodobnie najbardziej użyteczna jest możliwość skierowania, przy użyciu funkcji potoku Uniksa, standardowego wyjścia jednego programu na standardowe wejście innego programu. Następujące polecenie powłoki oznacza, że wszystko, co jest zapisywane przez `prog_1` na jego standardowym wyjściu, staje się standardowym wejściem dla `prog_2`:

```
$ prog_1 | prog_2
```

Te standardowe deskryptory plików wejściowych i wyjściowych oferują możliwość tworzenia elastycznych, spójnych programów. Program może być opracowany jako ogólne narzędzie, które może na przykład, jeśli jest taka potrzeba, akceptować wejście bezpośrednio od użytkownika, z pliku lub nawet wyjścia innego programu. W każdym przypadku program po prostu czyta ze standardowego wejścia używając deskryptora pliku 0, a końcowa decyzja o źródle wejścia jest pozostawiona aż do czasu uruchomienia.

2.2.2 Przykład io

Będąc nadzwyczaj prošym przykładem użycia standardowych deskryptorów pliku, program `io` używa funkcji systemowych `read` i `write` oraz deskryptorów pliku 0 i 1 do kopiowania ze swojego standardowego wejścia do swojego standardowego wyjścia. Jest to w istocie skrócona wersja programu `cat` Uniksa. Zauważ nieobecność jakichkolwiek wywołań funkcji `open` lub `creat`.

```
/* io -- kopiuje standardowe wejście do standardowego wyjścia */

#include <stdlib.h>
#include <unistd.h>

#define SIZE 512

main ()
{
    ssize_t nread;
    char buf[SIZE];

    while((nread = read(0, buf, SIZE)) > 0)
        write(1, buf, nread);
    exit (0);
}
```

Załóżmy, że ten program jest zawarty w pliku źródłowym `io.c` i skompilowany w celu otrzymania wykonywalnego pliku binarnego `io`:

```
$ cc -o io io.c
```

Jeśli program `io` zostanie teraz wywołany po prostu przez wpisanie jego nazwy, powinien czekać na wejście z terminala. Gdy użytkownik wprowadzi linie danych i wciśnie klawisz [Return] lub [Enter], `io` powinien po prostu powtórnie wyświetlić wprowadzoną linię, czyli zapisać wejściową linię do standardowego wyjścia. Rzeczywisty dialog może wyglądać jakoś tak:

\$ <code>io</code>	(użytkownik wpisał <code>io</code> , a następnie naciął [Return])
to jest linia 1	(użytkownik wpisał <code>to</code> , a następnie naciął [Return])
to jest linia 1	(<code>io</code> powtórnie wyświetlił linię)
•	
•	
•	

Po powtórznym wyświetleniu linii, i o będzie czekał na więcej danych wejściowych. Użytkownik może kontynuować wpisywanie bez końca, zaś i o będzie posłusznie powtarzał wyświetlanie każdej linii, gdy tylko zostanie wciśnięty [Return] lub [Enter]. Aby zakończyć program, użytkownik może samemu wprowadzić w linii znak końca pliku dla systemu. Jest to typowo ^D, to jest [Ctrl+D], wysyłane przez równoczesne wciśnięcie klawiszy [Ctrl] i [D]. Powyższe działanie powinno spowodować, że read zwróci 0, wskazując, że został osiągnięty koniec danych. Kompletny dialog może wyglądać tak:

```
$ io
to jest linia 1
to jest linia 1
to jest linia 2
to jest linia 2
<Ctrl-D>      (użytkownik wpisał [Ctrl+D])
$
```

Bardziej spostrzegawczy czytelnik może zauważyc, że i o nie zachowuje się tak, jak moglibyśmy oczekiwac. Zamiast czytać pełne 512 znaków, zanim je wydrukuję, jak sugeruje logika programu, drukuje on każdą linię, gdy tylko zostanie wciśnięty klawisz [Return]. Dzieje się tak, ponieważ read, gdy przyjmuje dane z terminala, zwykle powraca po każdym znaku nowego wiersza – jest to właściwość ułatwiająca interakcję. Dokładniej mówiąc, jest to prawdziwe tylko dla najczęstszych ustawień terminala. Terminali mogą być także ustawione w inne tryby, pozwalając na wejście pojedynczych znaków. Temat ten omówimy dokładniej w rozdziale 9.

Ponieważ i o używa standardowych deskryptorów pliku, może być używany w połączeniu z własnościami przekierowania i potokami powłoki. Na przykład polecenie:

```
$ io </etc/motd> message
```

spowoduje, że i o skopiuje zawartość komunikatu pliku dnia /etc/motd do pliku message, podczas gdy wiersz polecenia:

```
$ io </etc/motd> wc
```

spowoduje powiązanie standardowego wyjścia i o z narzędziem do liczenia słów wc Uniksa. Ponieważ standardowe wyjście i o będzie w rzeczywistości identyczne z zawartością /etc/motd, jest to dość niewygodny sposób liczenia słów, linii i znaków w pliku.

Ćwiczenie 2.15 Napisz wersję i o, która sprawdza, czy są jakieś argumenty wiersza polecenia. Jeśli istnieją, program powinien traktować każdy argument jako nazwę pliku i kopować zawartość każdego pliku do swojego standardowego wyjścia. Jeśli nie ma żadnych argumentów wiersza polecenia, wejście powinno być pobie-

rane ze standardowego wejścia. Jak powinien zachować się nowy program i o, jeśli nie może otworzyć pliku?

Ćwiczenie 2.16 Czasami dane w pliku powinny gromadzić się powoli, w długim czasie. Napisz wersję i o wywołującą procedurę watch, która będzie czytać do końca pliku ze standardowego wejścia, kopując dane do standardowego wyjścia. Kiedy osiągnie koniec swojego wejścia, procedura watch powinna zrobić 5 sekund przerwy. Następnie program powinien ponownie uruchomić czytanie ze swojego standardowego wejścia, by zobaczyć, czy nie przybyło więcej danych, bez powtórnego otwierania pliku lub zmiany położenia wskaźnika odczytu-zapisu. Aby proces pozostał uśpiony przez określony czas, możesz użyć standardowego odprogramu bibliotecznego sleep, który pobiera pojedynczy argument – liczbę całkowitą, określającą ilość sekund czekania. Na przykład:

```
sleep(5);
```

każe procesowi spać pięć sekund. Procedura watch jest podobna do programu o nazwie readslow, który można spotkać w niektórych wersjach Uniksa. Możesz także zobaczyć w swoim podręczniku pozycję opisującą opcję -f polecenia tail.

2.2.3 Użycie standardowego wyjścia komunikatów o błędach

Standardowe wyjście komunikatów o błędach jest raczej specjalnym deskryptorem pliku wyjściowego, który jest zwyczajowo zarezerwowany dla błędów i komunikatów ostrzeżeń. Pozwala programowi oddzielić komunikaty o błędach od normalnego wyjścia. Na przykład użycie standardowego wyjścia komunikatów o błędach pozwala programowi wyświetlać komunikaty błędów na terminalu, podczas gdy standardowe wyjście jest zapisywane do pliku. Jednak, jeśli jest to potrzebne, standardowe wyjście komunikatów o błędach może też być przekierowane w podobny sposób jak standardowe wyjście. Na przykład standardowe polecenie powłoki:

```
$ make > log.out 2> log.err
```

spowoduje wysłanie komunikatów o błędach polecenia make do pliku log.err. Standardowe wyjście jest wysyłane do log.out.

Programista może użyć w programie standardowego wyjścia komunikatów o błędach, stosując funkcję systemową write i deskryptor pliku 2:

```
char msg[6]={"boob\n";
```

```
.
```

```
write(2, msg, 5);
```

Jest to jednak toporne i niewygodne. Lepsze rozwiązanie podamy pod koniec tego rozdziału.

2.3 Przegląd Standardowej Biblioteki I/O

Funkcje systemowe dostępu do pliku dostarczają podstaw wszystkich operacji wejścia i wyjścia dla programów uniksowych. Jednak funkcje te wykonują operacje elementarne i obsługują dane tylko w formie prostej sekwencji bajtów, całą resztę pozostawiając programistie. Troska o efektywność także spada na barki projektanta.

Aby ułatwić życie, Unix oferuje Standardową Bibliotekę I/O ANSI C, zawierającą dużo więcej ułatwień, niż dotychczas opisane funkcje systemowe. Ponieważ nasza książka koncentruje się na interfejsie funkcji systemowych jądra, pełne omówienie Standardowej Biblioteki I/O odłożyliśmy aż do rozdziału 11. Jednak dla celów porównawczych warto już tu, w skrócie, zapoznać się z jej możliwościami.

Prawdopodobnie najbardziej oczywistą różnicą pomiędzy standardowym I/O a operacjami elementarnymi funkcji systemowych jest sposób opisu pliku. Zamiast deskryptora pliku jako liczby całkowitej, standardowe procedury I/O pracują domyślnie lub jawnie, ze strukturą o nazwie FILE. Następny przykład pokazuje, jak otworzyć plik za pomocą procedury fopen:

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

main()
{
    FILE *stream;

    if(( stream = fopen("junk", "r")) == NULL)
    {
        printf("Could not open file junk\n");
        exit (1);
    }
}
```

Pierwsza linia przykładu:

```
#include <stdio.h>

włacza plik nagłówkowy <stdio.h> Standardowej Biblioteki I/O. Ten plik zawiera, wśród wielu innych rzeczy, definicje FILE, NULL i deklaracje extern dla takich funkcji jak fopen. Obecnie NULL jest także zdefiniowany w <unistd.h>. Rzeczywista treść przykładu zawarta jest w instrukcji:
```

```
if(( stream = fopen("junk", "r")) == NULL)
{
    .
    .
}
```

gdzie junk jest nazwą pliku, podczas gdy napis "r" oznacza, że plik ma być otwarty tylko do czytania. Napis "w" powinien być użyty do obcięcia lub utworzenia pliku do zapisu. Jeśli wywołanie zakończy się pomyślnie, procedura fopen inicjuje strukturę FILE i zwraca jej adres przez stream. Wskaźnik stream może być następnie przekazywany do innych procedur bibliotecznych. Ważne jest zrozumienie, że gdzieś w ciele fopen dokonywane jest wywołanie naszej dobrej

znajomej, funkcji open. W następstwie tego gdzieś w strukturze FILE znajduje się deskryptor pliku, wiążący tę strukturę z plikiem. Istotne jest, że standardowe procedury I/O są napisane w oparciu o elementarne operacje funkcji systemowych. Główną funkcją biblioteki jest dostarczenie programistom bardziej przyjaznego interfejsu i automatycznego buforowania.

Gdy plik już zostanie otwarty, można skorzystać z wielu standardowych procedur I/O umożliwiających dostęp do niego. Jedną z nich jest getc, czytająca pojedynczy znak; inną jest putc, zapisującą pojedynczy znak. Są one używane w następujący sposób:

Użycie

```
#include <stdio.h>
int getc(FILE *istream);           /* czyta znak z istream */
int putc(int c, FILE *ostream);    /* umieszcza znak w ostream */
```

Obie procedury mogą być umieszczone w pętli do kopiowania jednego pliku do innego, jak poniżej:

```
int c;
FILE *istream, *ostream;

/* otwórz istream do odczytu i ostream do zapisu */
.
.
.
while(( c = getc(istream)) != EOF)
    putc(c, ostream);
```

Stała EOF jest zdefiniowana w <stdio.h> i zwracana przez procedurę getc, kiedy osiągnie ona koniec pliku. Rzeczywistą wartością EOF jest -1, dlatego wartość zwracana przez getc jest zdefiniowana jako int.

Na pierwszy rzut oka procedury getc i putc wyglądają niepokojąco, ponieważ przetwarzają pojedyncze znaki, a jak widzieliśmy, jest to wyjątkowo nieefektywne przy funkcjach systemowych. Standard I/O unika tej nieefektywności dzięki eleganckiemu mechanizmowi buforowania, który działa następująco: pierwsze wywołanie procedury getc powoduje odczytanie BUFSIZ znaków z pliku za pomocą funkcji systemowej read (jak widzieliśmy w podrozdziale 2.1.6, BUFSIZ jest stałą zdefiniowaną w <stdio.h>). Dane są przechowywane w buforze ustalonym przez bibliotekę (ale ciągle w przestrzeni adresowej użytkownika). Przez getc będzie zwrocony tylko pierwszy znak. Wszystkie pozostałe wewnętrzne działania są dobrze ukryte przed wywołującym programem. Kolejne wywołania procedury getc zwracają kolejne znaki z bufora. Gdy getc przekaże do programu BUFSIZE znaków, kolejne wywołanie getc spowoduje odczytanie następnej porcji znaków z pliku. Podobny mechanizm wyjściowy jest dostarczony dla procedury putc.

Ta technika jest bardzo użyteczna, ponieważ programista nie musi troszczyć się o efektywność. Jednak oznacza też, że dane są zapisywane dużymi fragmentami i zawartość pliku będzie znacznie opóźniona w stosunku do programu (specjalne ustalenia dotyczą tylko terminali). Dlatego niepożądane jest mieszanie standardowych procedur I/O i funkcji systemowych, jak `read`, `write` czy `lseek` w tym samym pliku. Jeśli nie wiesz dokładnie, co robisz, może powstać chaos. Z drugiej strony, można mieszać funkcje systemowe i standardowe procedury I/O, jeśli dotyczą one różnych plików.

Oprócz mechanizmu buforowania, standard I/O dostarcza narzędzi do formowania i konwersji; na przykład procedura `printf` oferuje formatowane wyjście:

```
printf("An integer %d\n", ival);
```

Powinno to być znane większości czytelników tej książki. (A tak przy okazji, `printf` niejawnie zapisuje do standardowego wyjścia).

Wydruk komunikatów o błędach za pomocą fprintf

Do wyświetlanego komunikatów diagnostycznych można wykorzystać procedurę `printf`. Niestety, pisze ona do standardowego wyjścia, a nie do standardowego wyjścia komunikatów o błędach. Jednak w tym celu możemy użyć `fprintf`, uogólnionej wersji `printf`. Następujący fragment programu pokazuje, jak to zrobić:

```
#include <stdio.h>           /* definicja stderr */
.
.
.
fprintf(stderr, "error number %d\n", errno);
```

Jedyna różnica pomiędzy tym użyciem procedury `fprintf` a wywołaniem funkcji `printf` to parametr `stderr`. Jest on wskaźnikiem do standardowej struktury `FILE`, automatycznie powiązanej ze standardowym wyjściem komunikatów o błędach.

Następująca procedura rozszerza użycie `fprintf` do procedury błędu o bardziej ogólnym zastosowaniu:

```
/* notfound -- drukuje błąd pliku i wychodzi */
#include <stdio.h>
#include <stdlib.h>

int notfound(const char *progname, const char *filename)
{
    fprintf(stderr, "%s: file %s not found\n", progname,
            filename);
    exit(1);
}
```

W późniejszych przykładach dla komunikatów o błędach będziemy używać `fprintf`, a nie `printf`. Zapewni to spójność z większością poleceń i programów, które dla diagnostyki używają standardowego wyjścia komunikatów o błędach.

2.4 Zmienna errno i funkcje systemowe

Jak widzieliśmy, wszystkie dotychczas opisane funkcje systemowe dostępu do pliku mogą w jakiś sposób zawieść. Jest to uniwersalnie wskazywane przez zwrót wartości -1. Aby pomóc programowi w uzyskaniu dokładniejszej informacji, kiedy taki wyjątek nastąpił, Unix dostarcza globalnie dostępnej zmienną całkowitą, zawierającą numer kodu błędu. Numer kodu błędu jest powiązany z takimi komunikatami o błędach jak „no permission” (brak pozwolenia) czy „invalid argument” (nieprawidłowy argument). Kompletna lista kodów błędów i ich znaczenia została zamieszczona w dodatku A. Numer błędu zawiera ostatni typ błędu, który wystąpił podczas funkcji systemowej.

Nazwą zmiennej błędu jest `errno`. Programista może używać `errno` wewnątrz programu w C po włączeniu pliku nagłówkowego `<errno.h>`. Większość starych programów utrzymuje, że zmienna całkowita `errno` jest deklarowana z zewnętrznym wiązaniem; XSI spowodowało, że ta deklaracja stała się przestarzała.

Następujący program wywołuje `open` i jeśli to zawiedzie, program powinien użyć `fprintf` do wyświetlenia wartości `errno`:

```
/* errl.c -- otwarcie pliku z obsługą błędów */

#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main ()
{
    int fd;

    if(( fd = open("nonesuch", O_RDONLY)) == -1)
        fprintf(stderr, "error %d\n", errno);
}
```

Jeżeli na przykład plik `nonesuch` nie istnieje, w standardowej realizacji Uniksa będzie wyświetlony kod błędu 2. Podobnie jak wszystkie możliwe wartości `errno`, ten kod jest także podany jako nazwa symboliczna (w tym przypadku `ENOENT`, co znaczy po prostu „no such file or directory” (nie ma takiego pliku lub katalogu)). Możesz wykorzystać bezpośrednio te nazwy symboliczne, tak jak są one zdefiniowane za pomocą dyrektywy `#define` w pliku nagłówkowym `<errno.h>`.

Trzeba uważać przy korzystaniu z `errno`, ponieważ nie jest on zerowany przy nowym wywołaniu funkcji systemowej. Dlatego najlepiej jest używać `errno` bezpośrednio po wywołaniu funkcji systemowej, której zawiodło.

2.4.1 Podprogram perror

Oprócz `errno`, Unix dostarcza procedurę biblioteczną (nie funkcję systemową) o nazwie `perror`. Dla większości tradycyjnych poleceń Uniksa jest to standardowy sposób relacjonowania błędów. Procedura ta pobiera pojedynczy argument w postaci napisu (cięgu znaków). Kiedy zostanie wywołana, tworzy na standar-

dowym wyjściu komunikatów o błędach komunikat zawierający przekazany do procedury argument napisowy, dwukropki i dodatkowy komunikat powiązany z bieżącą wartością zmiennej `errno`. Komunikat o błędzie jest drukowany na standardowym wyjściu komunikatów o błędach, a nie na standardowym wyjściu.

W powyższym przykładzie, linia zawierająca wywołanie funkcji `printf` może być zastąpiona przez:

```
perror("error opening nonesuch");
```

Jeśli plik `nonesuch` nie istnieje, wtedy `perror` powinna wyświetlić komunikat:

```
error opening nonesuch: No such file or directory
```

Ćwiczenie 2.17 Napisz procedury, które imitują opisane w tym rozdziale elementarne operacje dostępu do pliku, ale wywołują `perror`, kiedy wystąpi wyjątek lub błąd.

ROZDZIAŁ 3

Plik w kontekście

3.1 Pliki w środowisku wielu użytkowników

3.2 Pliki z wieloma nazwami

3.3 Uzyskiwanie informacji o pliku: stat i fstat

Pliki nie są w pełni określone przez zawarte w nich dane. Każdy plik uniksowy posiada też pewną liczbę dodatkowych własności pierwotnych, koniecznych dla administrowania nim w złożonym systemie z wieloma użytkownikami. To są te dodatkowe własności, a funkcje systemowe, które manipulują nimi, będąmy omawiać w tym rozdziale.

3.1 Pliki w środowisku wielu użytkowników

3.1.1 Użytkownicy i właściciele

Każdy plik w systemie Unix jest własnością jednego z użytkowników systemu. Zwykle jest to użytkownik, który utworzył plik. Tożsamość aktualnego właściciela reprezentuje nieujemna liczba całkowita nazywana **identyfikatorem użytkownika** (ang. *user-id*, często skracane do *uid*), doliczana do pliku w chwili jego tworzenia.

W typowym systemie Unix, identyfikator użytkownika powiązany z konkretną nazwą użytkownika, może być znaleziony w trzecim polu pozycji użytkownika w pliku hasel to znaczy w linii pliku `/etc/passwd`, identyfikującej użytkownika systemu. Typowa pozycja:

```
keith:x:35:10::/usr/keith:/bin/ksh
```

wskazuje, że użytkownik `keith` ma uid równe 35.

Pola w pozycjach pliku hasel są rozdzielane dwukropkami. Pierwsze pole zawiera nazwę użytkownika (ang. *username*). Drugie, reprezentowane tu przez `x`, jest znacznikiem miejsca dla hasła użytkownika. W odróżnieniu od wcześniejszych wersji Uniksa, zaszyfrowane aktualne hasło jest teraz najczęściej trzymane w innym pliku (ta implementacja zależy od systemu). Jak widzieliśmy, trzecie pole zawiera identyfikator użytkownika. Czwarte pole zawiera identyfikator domyślnej grupy użytkownika (ang. *group-id*, skracane do *gid*), który objaśnimy szczegółowo za

chwilę. Piąte to opcjonalne pole komentarza. Szóste podaje katalog macierzysty (ang. *home directory*) użytkownika. Ostatnie pole jest nazwą ścieżki programu, uruchamianego po zalogowaniu się użytkownika. Na przykład `/bin/ksh` jest jedną ze standardowych powłok Uniksa.

W Uniksie user-id jest rzeczywiście ważny przy identyfikacji użytkownika. Każdy proces Uniksa jest zwykle powiązany z uid użytkownika, który rozpoczął proces. (Pamiętaj, że proces to po prostu wykonywanie programu). Faktyczna nazwa użytkownika jest w istocie wygodną dla ludzi konwencją mnemoniczną. Kiedy plik jest tworzony, system ustala własność przez odniesienie do uid tworzącego proces.

Własność pliku może być później zmieniona, ale tylko przez uprzywilejowanego użytkownika (super-użytkownika, czyli administratora) systemu albo właściciela pliku. Warto zauważyć, że administrator ma nazwę użytkownika `root` i zawsze uid równy 0.

Podobnie jak indywidualni użytkownicy, pliki też są łączone z grupami (ang. *groups*). Grupa to po prostu arbitralny zbiór użytkowników, co daje nam prostą metodę kontrolowania projektów dotyczących kilku osób. Każdy użytkownik przynależy co najmniej do jednej grupy, a zwykle do wielu.

Grupy są definiowane w pliku `/etc/group`. Każda grupa jest identyfikowana przez `gid`, który, podobnie jak uid, jest nieujemną liczbą całkowitą. Domyślną grupę użytkownika wskazuje czwarte pole jego pozycji w pliku hasel.

Podobnie jak uid, gid użytkownika jest dziedziczony przez proces zapoczątkowany przez użytkownika. Tak więc, kiedy plik jest tworzony, gid powiązany z tworzącym go procesem będzie przechowywany razem z uid.

Efektywne identyfikatory użytkownika i grupy

Mówiąc dokładniej, powinniśmy opisywać tworzenie pliku w stosunku do efektywnego `user-id` (ang. *effective user-id*, często skracany do *euid*), powiązanego z procesem. Dzieje się tak, ponieważ proces może być rozpoczęły wprawdzie przez jednego użytkownika (na przykład `keith`), ale w pewnych specyficznych sytuacjach może nabyć przywileje systemu plików innego użytkownika (na przykład `diny`). Niebawem zobaczymy, jak to się dzieje. Uid użytkownika, który faktycznie zapoczątkował proces, jest opisany jak rzeczywisty identyfikator użytkownika (ang. *real user-id*, często także skracany do *ruid*) tego procesu. Oczywiście w większości przypadków efektywny i rzeczywisty identyfikator użytkownika są zbieżne.

Z podobnych przyczyn istnieje także efektywny identyfikator grupy (ang. *effective group-id*, skracany do *egid*) procesu, który ustala grupę powiązaną z plikiem.

3.1.2 Uprawnienia i tryby pliku

Podobnie jak każdy rodzaj własności, własność pliku daje pewne przywileje właścielowi. W szczególności właściciel może wybierać powiązane z plikiem uprawnienia (ang. *permissions*), nazywane też prawami dostępu. Uprawnienia określają, jak różni użytkownicy mogą uzyskać dostęp do pliku. Istnieją trzy typy użytkowników:

Rozdział 3: Plik w kontekście

- właściciel pliku
- każdy, kto należy do tej samej grupy, co grupa powiązana z plikiem. Zauważ, że o ile właściciel jest zainteresowany, uprawnienia dla kategorii 1 przesłaniają uprawnienia dla grupy pliku.
- każdy, kto nie należy do kategorii 1 albo 2.

Dla każdej kategorii użytkowników istnieją trzy podstawowe typy praw dostępu do pliku. Określają one, co wolno robić użytkownikowi konkretnej kategorii:

- odczytywać z pliku;
- zapisywać do pliku;
- wykonywać plik (w tym przypadku plik zwykle zawiera program albo listę poleceń powłoki).

Jak zwykle super-użytkownik (administrator) stanowi szczególny przypadek i może manipulować każdym plikiem nie zważając na powiązane z nim prawa odczytu, zapisu czy wykonywania.

System przechowuje uprawnienia powiązane z plikiem jako wzorzec bitowy nazywany trybem pliku (ang. *file mode*). Chociaż plik nagłówkowy `<sys/stat.h>` zawiera symboliczne nazwy bitów uprawnień, większość programistów Uniksa ciągle woli używać stałych ósemkowych, pokazanych w tabeli 3.1 – nazwy symboliczne stenowią najnowszą – niezbyt zgrabną innowację. Pamiętaj, że w C stale ósemkowe są zawsze poprzedzane wiadącym 0, w przeciwnym przypadku kompilator potraktuje je jako stałe dziesięcinnie.

Tabela 3.1 Wartości ósemkowe dla tworzenia uprawnień pliku

Wartość ósemkowa	nazwa symboliczna	znaczenie
0400	S_IRUSR	pozwolenie odczytu przez użytkownika
0200	S_IWUSR	pozwolenie zapisu przez użytkownika
0100	S_IXUSR	właściciel może uruchamiać plik
0040	S_IRGRP	pozwolenie odczytu przez grupę
0020	S_IWGRP	pozwolenie zapisu przez grupę
0010	S_IXGRP	członek grupy może uruchamiać plik
0004	S_IROTH	inne użytkownicy mogą czytać z pliku
0002	S_IWOTH	inne użytkownicy mogą zapisywać do pliku
0001	S_IXOTH	inne użytkownicy mogą uruchamiać plik

Na podstawie tabeli łatwo jest wywnioskować, że możemy uczynić plik dostępnym do odczytu przez wszystkich użytkowników przez dodanie 0400 (pozwolenie odczytu przez właściciela), 040 (pozwolenie odczytu przez grupę) i 04 (pozwolenie odczytu dla wszystkich innych użytkowników). To daje końcowy tryb pliku 0444. Tryb może być też określony przez wykonanie alternatywy bitowej OR (+) na reprezentacjach symbolicznych, na przykład 0444 jest równeżne z:

`S_IRUSR | S_IWGRP | S_IROTH`

Ponieważ żadna inna wartość z tabeli nie została włączona, ten konkretny tryb 0444 oznacza też, że żaden użytkownik, włącznie z właścicielem, nie może zapisywać do pliku ani go uruchomić.

Aby to obejść, można użyć więcej niż jednej wartości ósemkowej dla każdej kategorii użytkowników. Na przykład dodając 0400, 0200 i 0100 otrzymamy wartość 0700 co oznacza, że właściciel może czytać z pliku, zapisywać do niego i uruchamiać plik.

Dlatego najbardziej prawdopodobna wartość trybu:

$$0700 + 050 + 05 = 0755$$

oznacza, że właściciel może odczytywać, zapisywać lub uruchamiać plik, podczas gdy członkowie grupy powiązanej z plikiem i każdy inny typ użytkowników jest ograniczony tylko do odczytu lub wykonywania pliku.

Zrozumiałe, że programiści Uniksa wolą używać stałych ósemkowych zamiast nazw z pliku `<sys/stat.h>`, gdzie prosta wartość 0755 jest reprezentowana przez:

`S_ISUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IXOTH`

To nie jest koniec historii uprawnień. W następnym podrozdziale zobaczymy, jak trzy inne typy uprawnień oddziałują na pliki zawierające programy wykonywalne. Być może ważniejsze niż dostęp do pliku jest to, że każdy katalog Uniksa ma też zestaw praw dostępu, zupełnie jak zwykły plik; dotyczy on dostępności pliku w katalogu. Szczegółowo będziemy omawiać to zagadnienie w rozdziale 4.

Ćwiczenie 3.1 Co oznaczają następujące wartości uprawnień: 0761, 0777, 0555, 0007 i 0707?

Ćwiczenie 3.2 Przetłumacz wartości ósemkowe z ćwiczenia 3.1 na ich ekwiwalenty symboliczne.

Ćwiczenie 3.3 Napisz procedurę `1soct`, która tłumaczy zestaw uprawnień, z postaci określonej przez `ls` (np. `rwxr-xr-x`) na ekwiwalent ósemkowy. Następnie napisz procedurę odwrotną `oct1s`.

3.1.3 Dodatkowe uprawnienia dla plików wykonywalnych

Istnieją trzy inne typy uprawnień plikowych, określane przez specjalne atrybuty i zwykle mające związek z plikami, zawierającymi program wykonywalny. Wartości ósemkowe i nazwy symboliczne, odpowiadające określonym bitom w trybie pliku, mają następujące znaczenie:

04000	<code>S_ISUID</code>	ustawia user-id przy wykonaniu
02000	<code>S_ISGID</code>	ustawia group-id przy wykonaniu
01000	<code>S_ISVTX</code>	save-text-image (bit klejący)

Jeśli uprawnienie `S_ISUID` jest ustawione, to podczas uruchamiania programu zawartego w pliku, system nadaje wynikowemu procesowi efektywny user-id,

pochodzący od *właściciela pliku*, a nie od użytkownika, który rozpoczął proces (normalnym stanem jest ten drugi przypadek). Tak więc proces przyjmuje przywileje systemu plików właściciela pliku, a nie użytkownika, który rozpoczął proces.

Za pomocą tego mechanizmu można kontrolować dostęp do ważnych danych; informacje mogą być chronione przed widokiem publicznym lub manipulacją za pomocą użycia standardowych praw odczytu-zapisu-wykonania. Właściciel pliku może więc utworzyć program, który uzyskuje dostęp do pliku w specyficzny, mocno zdefiniowany sposób. Kiedy program jest kompletny, może być ustawione jego uprawnienie `S_ISUID`, pozwalające innym użytkownikom na dostęp do pliku tylko za pomocą tego programu. Oczywiście, program trzeba pisać uważnie, tak, aby uniknąć jakiegokolwiek czasowego nadużycia przywileju.

Klasycznym przykładem tej techniki jest program `passwd`. Administrator systemu wprost prosi się o kłopoty, jeżeli zezwala, aby każdy użytkownik zapisywał do pliku haseł, gdy tylko przyjdzie mu na to ochota. Jednak wszyscy użytkownicy muszą czasami zapisać do tego pliku, aby zmienić swoje hasło. Program `passwd` omija ten problem, ponieważ jest własnością administratora i ma ustawiony bit `S_ISUID`.

Prawdopodobnie jest to mniej użyteczne, jednak uprawnienie `S_ISGID` robi to samo dla group-id pliku. Jeśli zostało ustawione w trakcie uruchamiania pliku, wynikowy proces nabędzie gid właściciela pliku, a nie użytkownika, który uruchomił program.

Dawniej bit `S_ISVTX` był ustawiany w plikach wykonywalnych i znany jako uprawnienie `save-text-image` (zachowaj obraz tekstu) albo częściej jako bit klejący (ang. *sticky bit*). We wczesnych systemach, jeśli w pliku był ustawiony bit `save-text-image`, w czasie wykonywania pliku tekst (kod) jego programu pozostawał w systemowym obszarze wymiany, dopóki system nie został zatrzymany. Tak więc, gdy program był wywoływanego po raz kolejny, system nie poszukiwał go w strukturze katalogu systemowego, ale po prostu (i szybko) przerzucał do pamięci. We współczesnych systemach Unix ten bit jest teraz nadmiarowy. XSI definiuje bit `S_ISVTX` tylko dla katalogów. Użycie bitu `S_ISVTX` będzie dokładniej wyjaśnione w rozdziale 4.

Ćwiczenie 3.4 Następujące przykłady pokazują, jak `ls` odpowiednio wyświetla ustawione uprawnienia user-id i group-id:

```
r-sr-xr-x  
r-xr-sr-x
```

Stosując polecenie `ls -l`, sprawdź zawartość katalogów `/bin`, `/etc` i `/usr/bin`, szukając pliku z niezwykłymi uprawnieniami tego typu. Bardziej doświadczeni czytelnicy mogą przyspieszyć poszukiwanie, używając programu `grep`. Jeśli znajesz jakiś plik z dziwnym uprawnieniem, wyjaśnij, dlaczego.

3.1.4 Maska tworzenia pliku i funkcja systemowa umask

Jak podaliśmy w skrócie w rozdziale 2, początkowe uprawnienia pliku są ustawiane w czasie tworzenia pliku za pomocą wywołania funkcji `creat` lub `open` w ich rozszerzonej formie, na przykład:

```
filedes = open("datafile", O_CREAT, 0644);
```

Z każdym procesem powiązana jest wartość nazywana **maską tworzenia pliku** (ang. *file creation mask*). Używa się jej do automatycznego wyłączania bitów uprawnień w czasie tworzenia pliku, niezależnie od trybu podanego w odpowiednim wywołaniu `creat` lub `open`. Jest to użyteczne przy zabezpieczaniu wszystkich plików tworzonych podczas istnienia procesu, ponieważ chroni przed przypadkowym włączeniem określonych uprawnień.

Podstawowa idea jest prosta: jeśli w masce tworzenia pliku zostanie ustawiony bit uprawnienia, wtedy będzie on zawsze wyłączony podczas tworzenia pliku. Bitы w masce mogą być ustawiane za pomocą tych samych stałych ósemkowych, jak opisane poprzednio dla trybów pliku, chociaż mogą być używane tylko podstawowe uprawnienia odczytu, zapisu i wykonywania. Bardziej egzotyczne uprawnienia, jak `S_ISUID`, nie mają żadnego znaczenia w masce tworzenia pliku.

W terminologii programistycznej instrukcja:

```
filedes = open(pathname, O_CREAT, mode);
```

jest zatem faktycznie równoważna z:

```
filedes = open(pathname, O_CREAT, (~mask) & mode);
```

gdzie maska zawiera bieżącą wartość maski tworzenia pliku, `~` jest operatorem negacji bitowej w C, a `&` to operator koniunkcji bitowej AND. Na przykład, jeśli wartość maski jest równa $04+02+01=07$, przy tworzeniu pliku uprawnienia zwykłe wskazywane przez te wartości są wyłączane. Tak więc plik tworzony za pomocą instrukcji:

```
fd = open("/tmp/newfile", O_CREAT, 0644);
```

w rzeczywistości będzie miał nadany tryb 0640. Oznacza to, że właściciel pliku i członkowie grupy powiązanej z plikiem będą mogli używać pliku, ale inni użytkownicy będą mieli zabroniony jakikolwiek dostęp. Maska tworzenia pliku dla procesu może być zmieniona za pomocą funkcji systemowej `umask`.

Użycie

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

mode_t umask(mode_t newmask);
```

Oto przykład:

```
mode_t oldmask;
```

Rozdział 3: Plik w kontekście

```
oldmask = umask(022);
```

Wartość 022 blokuje uprawnienie zapisu, dawane komuś innemu niż właściciel pliku. Typ `mode_t` został opracowany specjalnie w celu zapamiętania informacji o trybie pliku, czyli uprawnień. Jest on zdefiniowany w pliku nagłówkowym `<sys/types.h>`, który z kolei jest włączony do `<sys/stat.h>`. Po wywołaniu `oldmask` zawiera poprzednią wartość maski.

Wynika z tego, że jeśli chcesz być absolutnie pewny, że pliki są tworzone z dokładnie takim trybem, jak podany w wywołaniu `open` lub `creat`, powinieneś najpierw wywołać `umask` z argumentem równym零. Ponieważ wszystkie bity uprawnień w masce tworzenia pliku będą teraz zerami, żaden z bitów w trybie pliku przekazywanym do `open` lub `creat` nie będzie maskowany. Następujący przykład wykorzystuje tę ideę w celu utworzenia pliku z gwarantowanym trybem, a następnie przywraca starą maskę tworzenia pliku. Zwraca on uzyskany przez `open` deskryptor pliku.

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

int specialcreat(const char *pathname, mode_t mode)
{
    mode_t oldu;
    int filedes;

    /* ustaw na zero maskę tworzenia pliku */
    if((oldu = umask(0)) == -1)
    {
        perror("saving old mask");
        return(-1);
    }

    /* utwórz plik */
    if((filedes = open(pathname, O_WRONLY|O_CREAT|O_EXCL, mode)) == -1)
        perror ("opening file");

    /* odtwórz stary tryb pliku, nawet jeśli open się nie powiodło */
    if(umask(oldu) == -1)
        perror("restoring old mask");

    /* zwróć deskryptor pliku */
    return filedes;
}
```

3.1.5 Funkcja open i uprawnienia pliku

Jeśli funkcja `open` jest używana w celu otwarcia do odczytu lub zapisu istniejącego pliku, wtedy system sprawdza uprawnienia pliku, aby określić, czy tryb wymaganego przez proces dostępu (tylko odczyt, tylko zapis lub odczyt-zapis) jest dozwolony. Jeśli nie ma zgody na dostęp, funkcja `open` powinna zwrócić -1, wskazując niepowodzenie, a `errno` będzie zawierać kod błędu `EACCES`, oznaczający „permission denied” (brak uprawnień).

Kiedy funkcja open jest używana do tworzenia pliku w swoim rozszerzonym trybie, określenie wskaźników O_CREAT, O_TRUNC i O_EXCL oznacza, że traktowanie istniejącego pliku jest bardzo elastyczne. Oto przykłady użycia funkcji open z uprawnieniami pliku:

```
filedes = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, 0600);
if (filedes < 0)
    /* error */
filedes = open(pathname, O_WRONLY | O_CREAT | O_EXCL, 0600);
```

W pierwszym przykładzie, jeśli plik istnieje, to będzie obcięty pod warunkiem, że uprawnienia pliku pozwalały wywołującemu procesowi na dostęp do zapisu. W drugim przykładzie, jeśli plik istnieje, wywołanie open zawiedzie bez względu na uprawnienia pliku, a errno będzie zawierać kod EEXIST.

Ćwiczenie 3.5

- (a) Załóżmy, że proces ma euid równy 100 i egid równy 200. Plik testfile jest własnością użytkownika 101 i ma gid równy 200. Dla każdego możliwego trybu dostępu (tylko odczyt, tylko zapis i odczyt-zapis) określ, czy wywołanie open zakończy się sukcesem, jeśli testfile ma następujące uprawnienia:

rwxr-xrw-	r-xrw-r--	rwx---x---	rwsrw-r--
--s--s--x	---rwx---	---r-x--x	

- (b) Co się zdarzy, jeśli proces ma także rzeczywisty user-id równy 101 i rzeczywisty group-id równy 201?

3.1.6 Określanie dostępności pliku za pomocą funkcji access

access jest użyteczną funkcją systemową, określającą, czy proces może uzyskać dostęp do pliku, zgodnie z *rzeczywistym* user-id procesu, a nie bieżącym *efektywnym* user-id. Dostarcza to następnego poziomu bezpieczeństwa w procesie, który ma wzmocnione uprawnienia za pomocą bitu S_ISUID.

Użycie

```
#include <unistd.h>
int access(const char *pathname, int amode);
```

Jak widzieliśmy, istnieje kilka sposobów dostępu do pliku, aby więc podać systemowi więcej informacji, parametr amode zawiera wartość wskazującą na żądaną metodę dostępu. Parametr ten może przybierać następujące wartości, zdefiniowane w pliku <unistd.h>:

R_OK	Czy proces wywołujący ma dostęp do odczytu?
W_OK	Czy proces wywołujący ma dostęp do zapisu?
X_OK	Czy proces wywołujący może wykonać plik?

Jest tu tylko jeden zestaw wartości dla amode, ponieważ interesuje nas tylko możliwość dostępu do pliku dla jednego użytkownika, identyfikowanego przez ruid procesu wywołującego. Funkcja amode może też przyjmować wartość F_OK. Powoduje to wyłącznie sprawdzenie istnienia pliku.

Parametr pathname zawiera, jak zwykle, nazwę pliku.

Wartość zwracana przez funkcję access wynosi 0 (wskazując, że użytkownik identyfikowany przez ruid procesu może uzyskać dostęp do pliku w sposób wskazany przez amode) lub -1 (wskazując, że proces nie może uzyskać dostępu). W tym drugim przypadku, errno będzie zawierać wartość wskazującą przyczynę. Na przykład wartość EACCES oznacza, że uprawnienia pliku nie pozwalały na żądany dostęp, podczas gdy ENOENT oznacza, że plik po prostu nie istnieje.

Następujący zarys programu używa funkcji access w celu sprawdzenia, czy jego użytkownik może czytać plik, niezależnie od ustawienia bitu S_ISUID:

```
/* przykład użycia funkcji access */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main()
{
    char *filename = "afile";

    if(access(filename, R_OK) == -1)
    {
        fprintf(stderr, "User cannot read file %s\n", filename);
        exit(1);
    }

    printf ("%s readable, proceeding\n", filename);

    /* reszta programu ... */
}
```

Ćwiczenie 3.6 Napisz program whatable, który określi, czy możesz czytać, zapisywać lub uruchamiać plik. Kiedy typ dostępu nie jest dozwolony, wtedy whatable powinien określić, dlaczego (użyj perror).

3.1.7 Zmiana uprawnień pliku za pomocą funkcji chmod

Użycie

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

int chmod(const char *pathname, mode_t newmode);
```

Funkcja systemowa chmod służy do zmiany uprawnień istniejącego pliku. Może być używana w pliku tylko przez właściciela pliku lub administratora.

Parametr pathname wskazuje nazwę pliku. Parametr newmode zawiera nowy tryb pliku, zbudowany w sposób opisany w pierwszej części tego rozdziału.

Oto przykład użycia funkcji chmod:

```
if (chmod(pathname, 0644) == -1)
    perror("call to chmod failed");
```

Ćwiczenie 3.7 Napisz program setperm, który pobiera dwa argumenty z wiersza poleceń. Pierwszy jest nazwą pliku, drugi określeniem ósemkowym albo w stylu ls zestawu uprawnień. Jeśli plik istnieje, wtedy setperm powinien spróbować ustawić uprawnienia pliku na podaną wartość. Użyj procedury lsoct, którą opracowałeś w ćwiczeniu 3.3.

3.1.8 Zmiana właściciela za pomocą funkcji chown

Funkcja chown jest używana do zmiany właściciela i grupy pliku.

Użycie

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

int chown(const char *pathname, uid_t owner_id, gid_t group_id);
```

Na przykład:

```
int retval;
.

.

retval = chown("/usr/dina", 56, 3);
```

Jak widać, funkcja chown ma trzy argumenty: pathname, wskazujący nazwę ścieżki pliku, owner_id, wskazujący nowego właściciela i group_id, podający nową grupę. Wartość zwracana w retval wynosi 0 w przypadku sukcesu lub -1 w przypadku błędu. Typy uid_t i gid_t są zdefiniowane w pliku nagłówkowym <sys/types.h>.

W systemach zgodnych z XSI proces wywołujący musi być procesem super-użytkownika lub musi być właścicielem pliku i dlatego efektywny user-id procesu wywołującego musi być zgodny z właścicielem pliku. Przy jakiekolwiek nielegalnej próbie zmiany własności pliku zwracany jest błąd EPERM.

Ponieważ funkcja chown może być używana przez bieżącego właściciela pliku lub super-użytkownika, możliwe jest przekazanie komuś plików przez zwykłego użytkownika. Jednak ta operacja nie może być anulowana przez użytkownika

wykonującego wywołanie, ponieważ id użytkownika i nowy uid pliku już nie będą zgodne! Zauważ też, że celem zapobiegania pozbawionemu skrupułów użyciu funkcji chown do kradzieży przywilejów systemu plików, dla pliku, którego właściciel został zmieniony, wyłączone są pozwolenia ustawiania user-id i group-id. (Co mogłoby się zdarzyć, gdyby tak nie było?)

3.2 Pliki z wieloma nazwami

Każdy plik uniksowy może być identyfikowany przez więcej niż jedną nazwę. Inaczej mówiąc, ten sam fizyczny zbiór danych może być powiązany z kilkoma nazwami ścieżek Uniksa bez potrzeby dublowania pliku. Na początku może się to wydawać dziwne, ale jest bardzo użyteczne w warunkach oszczędzania miejsca na dysku albo zapewnienia pewnej liczbie osób możliwości użycia tego samego pliku.

Każda taka nazwa nosi miano łącza twardego (ang. *hard link*). Liczba łączyskojarzonych z plikiem jest nazywana licznikiem łączy (ang. *link count*) dla tego pliku.

Nowe łącze twarde jest tworzone za pomocą funkcji systemowej link, a istniejące łącze twarde może być usunięte za pomocą funkcji systemowej unlink.

3.2.1 Funkcja systemowa link

Użycie

```
#include <unistd.h>
int link(const char *original_path, const char *new_path)
```

Pierwszy parametr, original_path, jest wskaźnikiem znakowym, wskazującym nazwę ścieżki w Uniksie. Musi on identyfikować istniejące łącze do pliku, to znaczy istniejącą nazwę pliku. Drugi parametr, new_path, wskazuje nową nazwę albo łącze do pliku. Zauważ, że new_path nie musi istnieć jako plik.

Wywołanie funkcji systemowej link zwróci 0, jeśli jest pomyślne, albo -1, jeśli zdarzył się błąd. W tym drugim przypadku nie będzie utworzone żadne nowe łącze.

Na przykład instrukcja:

```
link("/usr/keith/chap.2", "/usr/ben/2.chap");
powinna utworzyć nowe łącze o nazwie /usr/ben/2.chap do istniejącego pliku /usr/keith/chap.2. Do pliku można się teraz odnosić za pomocą każdej z tych nazw. Jak widać, łącza nie muszą być w tym samym katalogu.
```

3.2.2 Ponownie funkcja systemowa unlink

W podrozdziale 2.1.13 wprowadziliśmy funkcję systemową `unlink` jako prosty sposób usuwania pliku z systemu. Na przykład:

```
unlink ("~/tmp/scratch");
```

`usunie ~/tmp/scratch.`

W rzeczywistości funkcja systemowa `unlink` usuwa tylko nazwę łącza i zmniejsza licznik łączy o jeden. Tylko jeśli licznik łączy został zmniejszony do zera i żaden program nie ma tego pliku aktualnie otwartego, dane z pliku będą nieodwracalnie usunięte z systemu. W tym przypadku poprzednio przydzielone plikowi bloki dyskowe są dodawane do utrzymywanej przez system listy wolnych bloków. Chociaż dane mogą jeszcze jakiś czas fizycznie istnieć, są już nie do odzyskania. Ponieważ większość plików ma tylko jedno łącze, jest to najczęstszy rezultat wywołania funkcji `unlink`. Natomiast, jeśli licznik łączy nie został zmniejszony do zera, dane pliku są pozostawione nietknięte i mogą być dostępne za pomocą innych łączy pliku.

Następujący krótki program zmienia nazwę pliku przez utworzenie najpierw łącza do nowej nazwy ścieżki i (jeśli jest to wykonane pomyślnie) usunięcia łącza do starej nazwy ścieżki. Jest to uproszczona wersja standardowego polecenia `mv` Uniksa.

```
/* move -- przesuwa plik z jednej nazwy ścieżki do innej */
```

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

char *usage = "usage: move file1 file2\n";

/* funkcja main używa argumentów wiersza poleceń,
 * przekazywanych w standardowy sposób
 */

main(int argc, char **argv)
{
    if(argc != 3)
    {
        fprintf(stderr, usage);
        exit(1);
    }

    if(link(argv[1], argv[2]) == -1)
    {
        perror("link failed");
        exit(1);
    }

    if(unlink(argv[1]) == -1)
    {
        perror("unlink failed");
        unlink(argv[2]);
        exit(1);
    }
}
```

```
}

printf("Succeeded\n");
exit (0);
}
```

Teraz jeszcze jedna końcowa kwestia. Dotychczas nie wspomnieliśmy o interakcji funkcji `unlink` i uprawnień związanych z jej argumentem w postaci nazwy pliku. A to dlatego, że po prostu uprawnienia te nie dotyczą funkcji `unlink`. Sukces albo niepowodzenie wywołania funkcji `unlink` zależy natomiast od uprawnień do *katalogu*, zawierającego plik. Temat ten będziemy szerzej omawiać w rozdziale 4.

3.2.3 Funkcja systemowa rename

W praktyce poprzedni przykład może być łatwiej wykonany za pomocą funkcji systemowej `rename`, która została dość niedawno dodana do Uniksa. Funkcja systemowa `rename` może być używana zarówno do zmiany nazwy katalogów, jak i zwykłych plików.

Użycie

```
#include <stdio.h>
int rename(const char *oldpathname, const char *newpathname);
```

Argument `oldpathname` wskazuje starą nazwę, zmienianą na nazwę wskazywaną przez drugi argument `newpathname`. Jeśli `newpathname` już istnieje, jest usuwany, zanim nazwa `oldpathname` zostanie zmieniona.

Ćwiczenie 3.8 Używając `unlink` napisz własną wersję polecenia `rm`. Twój program powinien sprawdzać za pomocą funkcji `access`, czy użytkownik ma uprawnienia zapisu do pliku. Jeśli ich nie ma, program powinien prosić o potwierdzenie przed próbą usunięcia pliku. (Dlaczego)? Bądź uważny przy testowaniu go!

3.2.4 Łącza symboliczne

Istnieją dwa ważne ograniczenia użycia funkcji `link`. Zwykły użytkownik nie może utworzyć łącza do katalogu i żaden użytkownik nie może utworzyć łącza do pliku przez różne systemy plików (ang. *file systems*). Systemy plików są zasadniczym składnikiem całej struktury plików Uniksa i będą omawiane szczegółowo w rozdziale 4.

Aby przezwyciężyć te ograniczenia, XSI obsługuje pojęcie **łączów symbolicznych** (ang. *symbolic links*). Łącze symboliczne stanowi faktycznie plik z jego własnymi prawami, ale zamiast zwykłych danych pliku zawiera ścieżkę pliku, do którego jest łączem. Można więc powiedzieć, że łącze symboliczne to wskaźnik do innego pliku.

Aby utworzyć łącze symboliczne, używamy funkcji systemowej `symlink`.

Użycie

```
#include <unistd.h>
int symlink(const char *realname, const char *symname);
```

Po ukończeniu funkcji `symlink` tworzony jest plik `symname`, wskazujący na plik `realname`. Jeśli wystąpi błąd (na przykład `symname` jest nazwą istniejącego pliku), wtedy funkcja `symlink` zwraca -1. W przeciwnym przypadku zwraca 0, oznaczające sukces.

Jeśli łącze symboliczne jest kiedykolwiek otwierane za pomocą funkcji `open`, funkcja systemowa `open` prawidłowo podąża ścieżką do `realname`. Jeśli jednak programista pragnie ujrzeć dane zawarte w samym `sysname`, musi użyć funkcji systemowej `readlink`.

Użycie

```
#include <unistd.h>
int readlink(const char *sympath, char *buffer, size_t bufsize);
```

Funkcja systemowa `readlink` najpierw otwiera `sympath`, następnie czyta zawartość pliku do `buffer` i na zakończenie zamknie `sympath`. Niestety *XSI* nie gwarantuje, że zawartość `buffer` będzie zakończona znakiem zerowym. Wartością zwracaną przez funkcję `readlink` jest liczba znaków w buforze lub -1 w przypadku błędu.

I jeszcze kilka słów ostrzeżenia na temat użycia łącz symbolicznych i podążania za nimi. Gdy zostanie usunięty oryginalny plik, wskazywany przez plik łącza symbolicznego, w przypadku próby uzyskania dostępu do tego pliku za pomocą łącza symbolicznego, zgłoszony zostanie nieco mylący błąd. Program będzie ciągle jeszcze „widział” łącze symboliczne, ale niestety funkcja `open` nie będzie mogła podążyć zawartą w nim ścieżką i powróci z `errno` ustawionym na `EEXIST`.

3.3 Uzyskiwanie informacji o pliku: stat i fstat

Dochodząc widzieliśmy tylko, jak ustawić lub zmienić podstawowe własności związane z plikiem. Dwie funkcje systemowe, `stat` i `fstat`, pozwalają procesowi poznać te własności dla istniejącego pliku.

Użycie

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *pathname, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

Rozdział 3: Plik w kontekście

Funkcja systemowa `stat` pobiera dwa argumenty: `pathname` jak zwykle wskazuje nazwę ścieżki, identyfikującej plik, podczas gdy drugi argument, `buf`, jest wskaźnikiem do struktury `stat`. Po pomyślnym wywołaniu funkcji ta struktura będzie zawierała informacje związane z plikiem.

Funkcja systemowa `fstat` jest prawie identyczna w działaniu z funkcją `stat`. Jedyną różnicą stanowi to, że zamiast nazwy ścieżki, `fstat` oczekuje deskryptora pliku. Dlatego funkcja `fstat` może być używana tylko z otwartymi plikami.

```
.
.
.
struct stat s;
int filedes, retval;

filedes = open ("/tmp/dina", O_RDWR);

/* struktura s może być teraz wypełniona za pomocą funkcji stat ... */
retval = stat("/tmp/dina", &s);

/* ... lub funkcji fstat */
retval = fstat(filedes, &s);
```

Definicja struktury `stat` znajduje się w systemowym pliku nagłówkowym `<sys/stat.h>` i zawiera następujące składowe:

<code>dev_t</code>	<code>st_dev</code>
<code>ino_t</code>	<code>st_ino</code>
<code>mode_t</code>	<code>st_mode</code>
<code>nlink_t</code>	<code>st_nlink</code>
<code>uid_t</code>	<code>st_uid</code>
<code>gid_t</code>	<code>st_gid</code>
<code>dev_t</code>	<code>st_rdev</code>
<code>off_t</code>	<code>st_size</code>
<code>time_t</code>	<code>st_atime</code>
<code>time_t</code>	<code>st_mtime</code>
<code>time_t</code>	<code>st_ctime</code>
<code>long</code>	<code>st_blksize</code>
<code>long</code>	<code>st_blocks</code>

Typy danych używane przez strukturę `stat` zdefiniowane są w systemowym pliku nagłówkowym `<sys/types.h>`.

Składowe struktury `stat` mają następujące znaczenie:

1. `st_dev`, `st_ino` Pierwsza z tych składowych struktury opisuje urządzenie logiczne, na którym znajduje się plik, a druga podaje *numer i-węzła* (ang. *inode number*) dla pliku. Ta liczba, w połączeniu ze `st_dev`, identyfikuje jednoznacznie plik. W rzeczywistości `st_dev` i `st_ino` sprawiają kłopoty podczas podstawowego zarządzania strukturą plików Uniksa. Objaśnimy te pojęcia w następnym rozdziale. Na razie możesz je spokojnie zignorować.
2. `st_mode` Zawiera tryb pliku i umożliwia programistom obliczenie uprawnień związanych z plikiem. Jeszcze słowo ostrzeżenia: wartość `st_mode` zawiera też informację o typie pliku i tylko najmłodsze 12 bitów jest związanych z uprawnieniami. Wyjaśnimy to w rozdziale 4.

3. **st_nlink** Liczba łączy nie symbolicznych (inaczej mówiąc liczba różnych nazw ścieżek) związanych z plikiem. Ta wartość jest aktualizowana przy każdym wywołaniu funkcji systemowych link i unlink.
4. **st_uid, st_gid** Wartości uid i gid dla pliku. Początkowo ustawiane przez funkcję creat lub open i zmieniane przez funkcję systemową chown.
5. **st_rdev** Ta składowa ma znaczenie tylko wtedy, gdy pozycja pliku jest używana do opisu urządzenia. Na razie możesz bezpiecznie ją ignorować.
6. **st_size** Bieżąca *logiczna* wielkość pliku w bajtach. Powinieneś być świadomy, że kiedy plik jest przechowywany na dysku, może być umieszczony na fizycznych obszarach granicznych dysku i dlatego faktyczna fizyczna wielkość pliku może być większa niż jego wielkość logiczna. Składowa **st_size** jest zmieniana przy każdym zapisie na końcu pliku.
7. **st_atime** Zawiera czas ostatniego odczytu danych z pliku (chociaż funkcje creat lub open na początku też ustawiają tę wartość).
8. **st_mtime** Zawiera czas dowolnej modyfikacji danych w pliku i jest ustawiana przy każdym zapisie do pliku.
9. **st_ctime** Zawiera czas dowolnej zmiany informacji zwracanej w samej strukturze stat. Zmienia ją wywołanie funkcji systemowych link (składowa **st_nlink**), chmod (składowa **st_mode**) i write (składowa **st_mtime** i być może **st_size**).
10. **st_blksize** Zawiera specyficzną dla systemu plików wielkość bloku I/O dla pliku. W niektórych systemach plików może być różna dla różnych plików.
11. **st_blocks** Zawiera liczbę fizycznych bloków systemu plików, przyporządkowanych temu konkretnemu plikowi.

Poniższy przykład podprogramu **filedata** wyświetla szczegóły związane z plikiem identyfikowanym przez nazwę ścieżki. Drukowana informacja składa się z wielkości pliku, identyfikatorów użytkownika i grupy oraz uprawnień odczyt-zapis-wykonanie dla pliku.

Aby pomóc w tłumaczeniu uprawnień pliku na formę czytelną, jak forma dwuznana przez ls, użyliśmy tablicy krótkich liczb całkowitych (short int) **octarray**, która zawiera wartości podstawowych uprawnień, i tablicy znaków **perms**, zawierającej ich odpowiedniki znakowe.

```
/* filedata - wyświetla informację o pliku */

#include <stdio.h>
#include <sys/stat.h>

/*
 * używa octarray w celu określenia,
 * czy bit uprawnienia jest ustawiony
 */
static short octarray[9] = { 0400, 0200, 0100,
                            0040, 0020, 0010,
```

```
0004, 0002, 0001);

/* kody mnemoniczne uprawnień plikowych,
 * o długości 10 znaków (z końcowym zerem)
 */
static char perms[10] = "rwxrwxrwx";`
```

```
int filedata(const char *pathname)
{
    struct stat statbuf;
    char descrip[10];
    int j;

    if(stat(pathname, &statbuf) == -1)
    {
        fprintf(stderr, "Couldn't stat %s\n", pathname);
        return(-1);
    }

    /* przekształca uprawnienia w formę czytelną */

    for(j=0; j<9; j++)
    {
        /*
         * testuje, czy uprawnienie jest ustawione,
         * używając koniunkcji bitowej (AND)
         */
        if(statbuf.st_mode & octarray[j])
            descrip[j] = perms[j];
        else
            descrip[j] = '-';
    }
    descrip[9] = '\0'; /* upewniamy się, że to napis */

    /* wyświetla informację o pliku */

    printf("\nFile %s :\n", pathname);
    printf("Size %ld bytes\n", statbuf.st_size);
    printf("User-id %d, Group-id %d\n", statbuf.st_uid,
           statbuf.st_gid);
    printf("Permissions: %s\n", descrip);
    return(0);
}
```

Następujący program lookout stanowi bardziej przydatne narzędzie. Mając podaną listę nazw plików sprawdza co minutę, czy żaden plik z listy nie zmienił się. W tym celu monitoruje czas modyfikacji (**st_mtime**) każdego pliku. To narzędzie jest przeznaczone do pracy w tle.

```
/* lookout - drukuje komunikat, gdy plik się zmienił */

#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>

#define MFILE      10
```

```

void cmp(const char *, time_t);
struct stat sb;

main(int argc, char **argv)
{
    int j;
    time_t last_time[MFILE+1];
    if(argc < 2)
    {
        fprintf(stderr, "usage: lookout filename ... \n");
        exit(1);
    }
    if(--argc > MFILE)
    {
        fprintf(stderr, "lookout: too many filenames\n");
        exit(1);
    }
    /* inicjacja */
    for(j=1; j<=argc; j++)
    {
        if(stat(argv[j], &sb)==-1)
        {
            fprintf(stderr, "lookout: couldn't stat %s\n",
                    argv[j]);
            exit(1);
        }
        last_time[j] = sb.st_mtime;
    }
    /* pętla sprawdzająca zmiany plików */
    for(;;)
    {
        for(j=1; j<=argc; j++)
            cmp(argv[j], last_time[j]);
        /*
         * czeka 60 sekund
         * "sleep" jest standardowa
         * procedura biblioteczna Uniksa
         */
        sleep(60);
    }
}

void cmp(const char *name, time_t last)
{
    /* dopóki można czytać statystykę pliku,
     * sprawdź czas modyfikacji */
    if(stat(name, &sb)==-1 || sb.st_mtime != last)
    {
        fprintf(stderr, "%s changed\n", name);
        exit(0);
    }
}

```

Ćwiczenie 3.9 Napisz program, który monitoruje i rejestruje zmiany wielkości pliku przez jedną godzinę. Na zakończenie powinien on przedstawić prosty histogram, pokazujący wszystkie zmiany w tym czasie.

Ćwiczenie 3.10 Napisz program slowwatch, który okresowo monitoruje czas modyfikacji wymienionego pliku (nie powinien zawodzić, jeśli plik początkowo nie istnieje). Kiedy plik będzie zmieniany, slowwatch powinien kopiować go na jego standardowe wyjście. Jak możesz zapewnić (albo domyśleć się), że plik jest w pełni zaktualizowany przed kopiowaniem?

3.3.1 Ponownie funkcja systemowa chmod

Funkcje systemowe stat i fstat rozszerzają użycie funkcji chmod. Ponieważ można teraz uzyskać informację o trybie pliku, program może modyfikować jego uprawnienia, zamiast tylko bezwarunkowo ustawiać je ponownie.

Demonstruje to następujący program addx. Najpierw wywołuje on funkcję stat w celu uzyskania bieżącego trybu pliku wymienionego na liście argumentów programu. Jeśli wywołanie zakończy się pomyślnie, program modyfikuje istniejące uprawnienia, aby plik mógł być wykonywany przez właściciela. Może to być użyteczne, jeśli plik zawiera „skrypt powłoki”.

```

/* addx -- dodaje uprawnienie wykonywania pliku */

#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#define XPERM 0100 /* uprawnienie wykonywania przez właściciela */

main(int argc, char **argv)
{
    int k;
    struct stat statbuf;

    /* pętla dla wszystkich plików na liście argumentów */
    for(k=1; k<argc; k++)
    {
        /* uzyskaj bieżący tryb pliku */
        if(stat(argv[k], &statbuf)==-1)
        {
            fprintf(stderr, "adx: couldn't stat %s\n", argv[k]);
            continue;
        }

        /* spróbuj dodać uprawnienia wykonywania
         * za pomocą operatora alternatywy bitowej (OR) */

        statbuf.st_mode |= XPERM;
        if(chmod(argv[k], statbuf.st_mode) == -1)
            fprintf(stderr, "adx: couldn't change mode for %s\n",
                    argv[k]);

    } /* koniec pętli */
    exit(0);
}

```

Najbardziej interesujący jest tu sposób modyfikacji trybu pliku za pomocą użycia operatora koniunkcji bitowej OR. Zapewnia to, że bit opisywany przez XPERM jest ustawiony. W praktyce możemy wydłużyć tę instrukcję do:

```
statbuf.st_mode = (statbuf.st_mode) | XPERM ;
```

Używamy krótszej formy dla przejrzystości. Możemy także zamiast XPERM użyć zdefiniowanej w systemie nazwy S_IXUSR.

Ćwiczenie 3.11 Ten przykład jest zbyt złożony, jak na zadanie, które wykonuje. Napisz więc (jeśli wiesz jak) równoważny przykład używając powłoki.

Ćwiczenie 3.12 Korzystając z opisu technicznego w twoim podręczniku Uniksa, napisz własną wersję polecenia chmod.

ROZDZIAŁ 4

Katalogi, systemy plików i pliki specjalne

- 4.1 Wstęp
- 4.2 Katalogi od strony użytkownika
- 4.3 Implementacja katalogu
- 4.4 Programowanie i katalogi
- 4.5 Systemy plików Uniksa
- 4.6 Pliki urządzeń Uniksa

4.1 Wstęp

W dwóch poprzednich rozdziałach koncentrowaliśmy się na podstawowej części struktury plików Uniksa – zwykłym pliku. W tym rozdziale zbadamy inne części struktury plików, czyli:

- *Katalogi* Katalogi stanowią magazyny nazw plików i w rezultacie pozwalają użytkownikom na grupowanie pewnych zestawów plików. Pojęcie katalogu jest dobrze znane większości użytkowników Uniksa i wielu osobom, które przesiadły się z innych systemów operacyjnych. Jak zobaczymy, katalogi Uniksa mogą być zagnieździone, co nadaje strukturze plików formę hierarchiczną, zbliżoną do drzewa.
- *Systemy plików* Systemy plików są zbiorami katalogów i plików. Zawierają całe podsekcje hierarchicznego drzewa katalogów i plików, tworzących strukturę plików Uniksa. Zazwyczaj system plików odpowiada fizycznym częściom (partycom) dysku lub caemu dyskowi. W większości przypadków są one niewidoczne dla użytkownika.
- *Pliki specjalne* Unix rozszerza pojęcie pliku tak, że obejmuje ono również urządzenia zewnętrzne, dołączone do systemu: drukarki, dyski, a nawet pamięć. Są one reprezentowane w strukturze plików przez nazwy plików. Plik, reprezentujący w ten sposób urządzenie, nazywany jest plikiem specjalnym (ang. *special file*). Może on być dostępny za pośrednictwem systemowych

funkcji dostępu do plików, które omówiliśmy w rozdziałach 2 i 3 (na przykład `open`, `read` i `write`). Każde takie wywołanie uaktywnia zawarty w jądrze kod sterownika urządzenia, odpowiedzialny za kontrolowanie danego urządzenia. Jednak program nie musi nic o tym wiedzieć; system zapewnia, że pliki specjalne mogą być traktowane prawie tak samo jak zwykłe pliki.

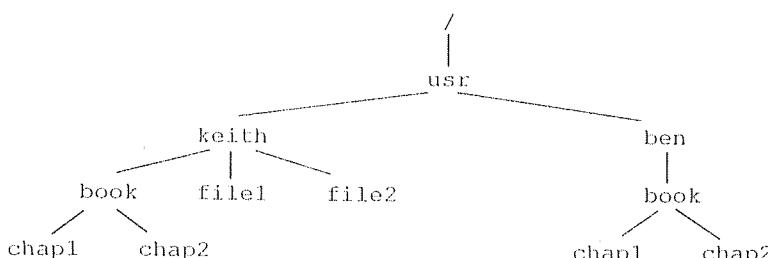
4.2 Katalogi od strony użytkownika

Nawet przypadkowy użytkownik Uniksa ma pewne pojęcie o wyglądzie jego struktury katalogów z poziomu polecień. Jednak dla porządku omówimy krótko sposób, w jaki użytkownik widzi układ plików i katalogów.

W gruncie rzeczy katalogi są tylko zbiorami nazw plików, co pozwala na logiczny podział plików na logicznie powiązane podgrupy. Na przykład każdy użytkownik zazwyczaj ma zapewniony swój własny **katalog macierzysty** (ang. *home directory*), do którego jest przenoszony po zarejestrowaniu się i gdzie może tworzyć i zmieniać pliki. Ma to oczywiście sens, ponieważ pozwala na trzymanie oddzielnie plików różnych użytkowników. Programy publiczne, takie jak `cat` i `ls`, są w podobny sposób trzymane razem w kilku katalogach, o nazwach takich jak `/bin` lub `/usr/bin`. Katalogi mogą być porównywane do szuflad w szafce na akta, używanych do grupowania papierowych teczek.

Katalogi są jednak wygodniejsze od szafek. Oprócz przechowywania plików, mogą one zawierać także inne katalogi, nazywane **podkatalogami** (ang. *subdirectories*), tworząc w ten sposób dodatkowe poziomy grupowania. Podkatalogi mogą z kolei zawierać swoje własne podkatalogi, a to zagnieżdżanie może mieć dowolną głębokość.

W istocie struktura plików Uniksa może być reprezentowana przez odwróconą, hierarchiczną strukturę podobną do drzewa. Uproszczone drzewo katalogu jest pokazane na rysunku 4.1 (jest to ten sam przykład, co w rozdziale 1). Oczywiście każdy rzeczywisty system może mieć bardziej złożony układ.



Rysunek 4.1. Przykład drzewa katalogu

Na górze naszego przykładowego drzewa i każdego innego drzewa katalogu w Uniksie znajduje się pojedynczy katalog nazywany **katalogiem głównym** (ang. *root directory*). Ma on raczej zwyczłą nazwę `/`. Niekońcowe węzły drzewa, jak `keith`

lub `ben`, są zawsze katalogami. Węzły końcowe, jak `file1` lub `file2`, są plikami zwykłymi, plikami specjalnymi albo pustymi katalogami. Większość systemów Unix pozwala obecnie na długość nazw katalogów do 255 znaków, ale podobnie jak w przypadku nazw plików, w celu zachowania przenośności powinny mieć co najwyżej 14 znaków.

W naszym przykładzie `keith` i `ben` są podkatalogami swojego katalogu rodzicielskiego (ang. *parent directory*) o nazwie `usr`. W katalogu o nazwie `keith` są trzy pozycje: dwa zwykłe pliki o nazwach `file1` i `file2` oraz podkatalog `book` o nazwie `book`. Z punktu widzenia podkatalogu `book`, jego katalogiem rodzicielskim jest `keith`. Z kolei `book` zawiera dwa pliki, `chap1` i `chap2`. Jak pokazano w rozdziale 1, położenie pliku w hierarchii może być określone za pomocą nazwy ścieżki. Na przykład pełna nazwa ścieżki dla pliku `chap2` znajdująca się w katalogu `keith` to `/usr/keith/book/chap2`. Katalogi również mogą być identyfikowane za pomocą nazw ścieżek. Nazwa ścieżki dla katalogu `ben` to `/usr/ben`.

Zauważ, że katalog `/usr/ben/book` także zawiera dwa pliki o nazwach `chap1` i `chap2`. Nie mają one żadnego związku ze swoimi imionami znajdująymi się w katalogu `/usr/keith/book`, ponieważ tylko pełna nazwa ścieżki niepowtarzalnie identyfikuje plik. Fakt, że pliki w innych katalogach mogą mieć te same nazwy, oznacza, że użytkownicy nie muszą ciągle wymyślać dziwnych, pięknych i unikalowych nazw plików.

Bieżący katalog roboczy

Zarejestrowany użytkownik pracuje w konkretnym miejscu struktury plików, nazywanym **biejącym katalogiem roboczym** lub czasami po prostu **katalogiem bieżącym**. Może nim być na przykład katalog, z którego polecenie `ls` będzie wyświetlało pliki, jeśli zostanie uruchomione bez argumentów. Początkowym ustaleniem bieżącego katalogu roboczego użytkownika jest jego katalog macierzysty, jak identyfikuje to systemowy plik hasel. Może być on zmieniony za pomocą polecenia `cd`. Na przykład:

```
$ cd/usr/keith
```

zmienia bieżący katalog roboczy na `/usr/keith`. Musisz wiedzieć, że polecenie `pwd` (nazwa `pwd` oznacza *print working directory* – wydrukuj katalog roboczy) wydrukuje nazwę bieżącego katalogu roboczego:

```
$ pwd
/usr/keith
```

Z punktu widzenia systemu, definicja bieżącego katalogu roboczego głosi, że jest to katalog, od którego system zaczyna poszukiwanie względnej nazwy ścieżki, to znaczy, że wyszukiwanie dotyczy nazwy ścieżki, która nie zaczyna się od `/`. Na przykład, jeśli bieżącym katalogiem roboczym jest `/usr/keith`, polecenie:

```
$ cat book/chap1
```

jest równoważne z:

```
$ cat /usr/keith/book/chap1
```

a polecenie:

`$ cat file1`

jest równoważne z:

`$ cat /usr/keith/file1`

4.3 Implementacja katalogu

W istocie katalogi Uniksa są po prostu plikami. System traktuje je prawie w ten sam sposób, jak zwykłe pliki. Posiadają właściciela, grupę, wielkość i powiązane prawa dostępu. Wielu z używanych do manipulacji plikami funkcji systemowych, omawianych w poprzednim rozdziale, można używać do manipulacji katalogami, chociaż nie jest to zalecane. Na przykład katalogi mogą być otwierane do odczytu za pomocą funkcji systemowej `open`, a zwrócony deskryptor pliku znajduje zastosowanie w kolejnych wywołaniach funkcji `read`, `lseek`, `fstat` i `close`.

Między katalogami i zwykłymi plikami istnieje jednak kilka ważnych różnic, narzuconych przez system. Katalogi nie mogą być utworzone za pomocą funkcji systemowych `creat` lub `open`. Również funkcja `open` nie będzie działać z katalogiem, który ma ustawiony znacznik `O_WRONLY` lub `O_RDWR`. Zamiast tego zawiedzie i ustawia `errno` na `EINVAL`. Te ograniczenia powodują, że niemożliwa jest aktualizacja katalogów za pomocą funkcji systemowej `write`. W istocie, z powodu specjalnej natury katalogów, znacznie lepiej jest używać specjalistycznej rodziny funkcji systemowych, które wkrótce omówimy.

Strukturalnie, katalogi składają się z szeregu pozycji katalogu, po jednej dla zawartego w nim każdego pliku albo podkatalogu. Każda pozycja katalogu zawiera co najmniej dodatkową liczbę nazywaną **numerem i-węzła** (ang. *inode number*) pliku i pole znakowe zawierające nazwę pliku. Dawniej, gdy nazwy plików były ograniczone do 14 znaków długości, pozycje katalogu miały ustaloną długość i większość systemów Uniksa używała tej samej metody implementacji (wyraźnym wyjątkiem był Berkeley UNIX). Jednak, gdy zostały wprowadzone długie nazwy plików, każda pozycja katalogu miała różną długość, co oznacza, że implementacje katalogu stały się zależne od systemu plików. Dlatego twoje programy nigdy nie powinny zakładać jakiegoś formatu katalogu i aby były naprawdę przenośne, powinny używać funkcji systemowych do manipulacji katalogami `XSI`.

Fragment katalogu zawierający trzy pliki może wyglądać jak na rysunku 4.2. (Nie pokazujemy tu żadnej informacji wymaganej do zarządzania wolną przestrzenią w katalogu). Katalog ten zawiera trzy pliki (które mogą być podkatalogami) o nazwach `fred`, `bookmark` i `abc`. Mają one odpowiednio numery i-węzłów 120, 207 i 235. Rysunek 4.2 przedstawia logiczną strukturę katalogu; w rzeczywistości katalog jest nieprzerwanym ciągiem bajtów.

120	f	r	e	d	\0
207	b	o	o	k	\0
235	a	b	c	\0	\0

Rysunek 4.2 Fragment katalogu

Numer i-węzła jednoznacznie identyfikuje plik (w rzeczywistości numery i-węzłów są unikatowe tylko w granicach systemu plików, ale powiemy o tym później). Numer i-węzła jest wykorzystywany przez system operacyjny do lokalizacji struktury danych opartej na dysku, nazywanej strukturą i-węzłów, która zawiera wszystkie informacje administracyjne pliku: wielkość, user-id właściciela, group-id, uprawnienia, czas ostatniego udostępnienia, czas ostatniej modyfikacji i adresy bloków na dysku, które zawierają dane pliku. Większość informacji dostarczanej przez opisywane w poprzednim rozdziale funkcje `stat` i `fstat` jest w rzeczywistości uzyskiwana bezpośrednio ze struktury i-węzłów. Bardziej szczegółowo omówimy tę strukturę w podrozdziale 4.5.

Ważne jest zrozumienie, że nasza reprezentacja katalogu to tylko jego obraz logiczny. Wydruk zawartości katalogu za pomocą polecenia `cat` może zakończyć się śmiertniakiem na ekranie. Lepszym sposobem badania katalogu jest użycie polecenia zrzutu ósemkowego (ang. *octal dump*) od z opcją `-c`. Na przykład, aby zobaczyć zawartość bieżącego katalogu roboczego, wypróbuj polecenie:

`$ od -c`

W tym poleceniu `.` (kropka) jest standardowym sposobem odnoszenia się do bieżącego katalogu roboczego.

4.3.1 Funkcje link i unlink raz jeszcze

W poprzednim rozdziale widzieliśmy, że funkcja systemowa `link` używana jest do tworzenia różnych nazw, odnoszących się do tego samego fizycznego pliku. Mechanizm jej działania powinien być teraz jasny. Każde łącze po prostu kończy się nową pozycją katalogu z tym samym numerem i-węzła, co dla oryginalnego pliku, ale z nową nazwą.

Jeśli w katalogu z rysunku 4.2 utworzymy łącze o nazwie `xyz` do pliku `abc` za pomocą następującego wywołania funkcji:

`link("abc", "xyz");`

nasz fragment katalogu może wyglądać jak na rysunku 4.3. Jeśli usuniemy łącze za pomocą funkcji systemowej `unlink`, odpowiednie bajty zawierające nazwę pliku zostaną zwolnione i mogą być powtórnie wykorzystane. Jeśli nazwa reprezentuje ostatnie łącze do pliku, cała struktura i-węzła jest zerowana. Powiązane bloki dyskowe, zawierające rzeczywiste dane pliku, są wtedy dodawane do utrzymywanej przez system listy wolnych bloków i stają się dostępne do ponownego użytku. W większości systemów Uniksa pliki nie mogą być nieusuwalne.

120	f	r	e	d	\0
207	b	o	o	k	m a r k \0
235	a	b	c	\0	
235	x	y	z	\0	

Rysunek 4.3 Przykładowy katalog z nowym plikiem

4.3.2 Kropka i podwójna kropka

W każdym katalogu zawsze są obecne dwie dziwne nazwy plików. Są to . i .. (kropka i podwójna kropka). Pojedyncza kropka jest standardową metodą Uniksa odnoszenia się do bieżącego katalogu roboczego, na przykład:

```
$ cat ./fred
```

co wyświetli plik fred z bieżącego katalogu, lub:

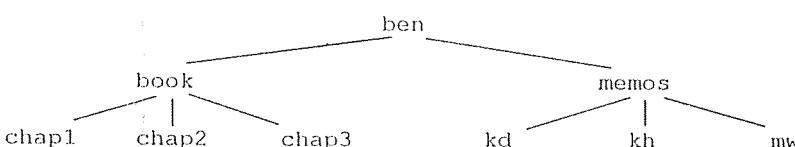
```
$ ls .
```

co pokaże wszystkie pliki w bieżącym katalogu. Podwójna kropka jest standardową metodą odnoszenia się do katalogu-rodzica (przodka) aktualnego katalogu roboczego; tj. katalogu, który zawiera aktualny katalog. Polecenie:

```
$ cd ..
```

przenosi więc użytkownika o jeden poziom wyżej w drzewie katalogów.

W rzeczywistości . i .. (kropka i podwójna kropka) są po prostu łączami odpowiednio do bieżącego katalogu roboczego i jego przodka, a każdy katalog Uniksa zawiera te dwie nazwy na swoich dwóch pierwszych pozycjach. Innymi słowy, każdy katalog w chwili utworzenia automatycznie zawiera te dwie nazwy. Możemy to zrozumieć, patrząc na przykładowy fragment drzewa katalogu, pokazany na rysunku 4.4.



Rysunek 4.4. Przykładowy fragment drzewa katalogu

Jeśli będziemy badać każdy z katalogów: ben, book i memos, zobaczymy coś takiego, jak na rysunku 4.5. Zauważ, że w katalogu book, . ma numer i-węzła równy 260, a .. numer i-węzła równy 123, co odpowiada pozycjom book i . w katalogu rodzicielskim ben. Podobnie . i .. w memos (401 i 123) odpowiadają memos i . w katalogu ben.

katalog ben

123	.	\0
247	.	\0
260	b	o
401	m	e
	m	o
	s	\0

katalog book

260	.	\0
123	.	\0
566	c	h
567	c	h
590	c	h
	a	p
	1	\0
	2	\0
	3	\0

katalog memos

401	.	\0
123	.	\0
800	k	h
810	k	d
077	m	w
		\0

Rysunek 4.5 Katalogi ben, book i memos

4.3.3 Prawa dostępu dotyczące katalogów

Podobnie jak zwykłe pliki, katalogi mają związane z nimi prawa dostępu, kontrolujące sposób, w jaki użytkownicy mogą uzyskać dostęp do nich.

Prawa dostępu do katalogów są organizowane dokładnie w ten sam sposób, co dla zwykłych plików, za pomocą trzech grup bitów 'rwx' określających przywileje właściciela katalogu, użytkowników w grupie właściciela i wszystkich innych użytkowników systemu. Jednak, chociaż te prawa dostępu są reprezentowane w dokładnie taki sam sposób, interpretuje się je inaczej.

- Prawo odczytu katalogu oznacza, że stosowna klasa użytkowników może wyświetlić nazwy zawartych w katalogu plików i podkatalogów. Nie znaczy to wcale, że użytkownicy mogą czytać informację zawartą w samych plikach – uprawnienia do tego są kontrolowane przez prawa dostępu indywidualnych plików.
- Prawo zapisu katalogu umożliwia użytkownikowi utworzenie nowego pliku i usunięcie istniejących plików z katalogu. I znowu, nie pozwala ono użytkownikowi modyfikować zawartości istniejącego pliku, dopóki nie pozwoli mu na to indywidualne uprawnienie pliku. Jednak możliwe jest usunięcie istniejącego pliku i utworzenie nowego z taką samą nazwą, co w rzeczywistości prowadzi do tego samego.
- Prawo wykonywania (nazywane też prawem przeszukiwania (ang. search permission)) katalogu pozwala użytkownikowi przejść do katalogu za pomocą polecenia cd lub funkcji systemowej chdir w programie (omówimy to

drugie poniżej). W dodatku, aby otworzyć plik lub wykonać program, użytkownik musi dysponować prawem wykonywania we wszystkich katalogach wiodących do pliku, określonych w bezwzględnej nazwie ścieżki pliku. Na poziomie poleceń powłoki, uprawnienia związane z katalogami mogą być zbadane przy użyciu opcji -l polecenia ls. Podkatalogi będą identyfikowane przez literę d na pozycji pierwszego znaku wydruku. Na przykład:

```
$ ls -l
total 168
-rw-r---- 1 ben other 39846 Oct 12 21:21 dir_t
drwxr-x--- 2 ben other 32 Oct 12 22:02 expenses
-rw-r---- 1 ben other 46245 Sep 13 10:34 new
-rw-r---- 1 ben other 3789 Sep 2 18:40 pwd_text
-rw-r---- 1 ben other 1310 Sep 13 10:38 test.c
```

Linia opisująca podkatalog expenses jest zaznaczona wiodącą literą d. Pokazuje ona, że katalog ma prawa dostępu odczytu, zapisu i wykonywania (przeszukiwania) dla swojego właściciela (użytkownika ben), prawa odczytu i wykonywania dla użytkowników w grupie pliku (o nazwie other) i żadnych praw dostępu dla wszystkich innych użytkowników.

Jeśli chcesz otrzymać wydruk informacji o bieżącym katalogu roboczym, przy poleceniu ls możesz dodać opcję -d (oprócz opcji -l). Na przykład:

```
$ ls -ld
drwxr-x--- 3 ben other 128 Oct 12 22:02 .
```

Pamiętaj, że nazwa . (kropka) na końcu wydruku używana jest w odniesieniu do bieżącego katalogu roboczego.

4.4 Programowanie i katalogi

Jak wspomniano wcześniej, istnieje specjalna rodzina funkcji obsługujących katalogi. Koncentrują się one głównie na strukturze typu dirent, która zdefiniowana jest w pliku nagłówkowym <dirent.h> i zawiera następujące składowe:

```
ino_t      d_ino;    /* numer i-węzła */
char       d_name[]; /* nazwa pliku, zakończona zerem */
```

Typ danych ino_t jest zdefiniowany w pliku nagłówkowym <sys/types.h>, który z kolei jest włączany w pliku nagłówkowym <dirent.h>. XSI nie określa wielkości d_name, ale gwarantuje, że ilość bajtów, poprzedzająca końcowe zero, będzie mniejsza niż liczba umieszczona w zmiennej _PC_NAME_MAX, zdefiniowanej w <unistd.h>. Zwrócić uwagę, że zerowa wartość d_ino oznacza pustą pozycję katalogu.

4.4.1 Tworzenie i usuwanie katalogów

Jak wspomniano wcześniej, katalogów nie można tworzyć używając funkcji systemowych creat lub open. Do tego celu przeznaczona jest specjalna funkcja systemowa mkdir.

Użycie

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
```

Pierwszy parametr pathname, wskazuje na ciąg znaków (napis), zawierający pełną nazwę katalogu, który ma być utworzony. Drugi parametr mode, jest zestawem praw dostępu dla tego katalogu. Uprawnienia te zostaną zmodyfikowane za pomocą wartości maski umask procesu, na przykład:

```
int retval;
retval = mkdir("/tmp/dir1", 0777);
```

Nie zaskoczy cię, że pomyślne wywołanie mkdir zwróci 0, a niepomyślne wywołanie -1. Co ważniejsze, powinieneś zauważyć, że mkdir umieści także dwa lącza, . i .. (kropka i dwie kropki) w utworzonym właśnie katalogu. Gdyby ich nie było, pozycja nie byłaby użyteczna jako katalog. Gdy katalog nie jest już potrzebny, może być usunięty za pomocą funkcji rmdir:

Użycie

```
#include <unistd.h>
int rmdir(const char *pathname);
```

Parametr pathname określa ścieżkę do katalogu, który ma być usunięty. Funkcja ta zadziała tylko wtedy, gdy katalog będzie pusty (to znaczy, gdy będzie zawierał tylko . (kropkę) i .. (dwie kropki)).

4.4.2 Otwieranie i zamknięcie katalogów

Aby otworzyć dowolny katalog Uniksa, XSI definiuje specjalną funkcję o nazwie opendir.

Użycie

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *dirname);
```

Parametrem przekazywanym do opendir jest ścieżka do katalogu, który ma być otwarty. Jeśli dirname zostanie pomyślnie otworzony, opendir zwróci wskaźnik do typu DIR. Plik nagłówkowy <dirent.h> zawiera definicję typu DIR, który reprezentuje strumień katalogu. Działa on w podobny sposób do typu FILE,

używanego w Standardowej Bibliotece I/O, opisanego w rozdziałach 2 i 11. Wskaźnik do strumienia katalogu jest ustawiany na pierwszą pozycję w katalogu. Jeśli wywołanie `opendir` nie powiedzie się, system zwróci wskaźnik pusty (ang. *null pointer*). Powinieneś zawsze pisać odpowiedni kod sprawdzający błędy, testując obecność pustego wskaźnika, zanim spróbujesz czytać ze strumienia katalogu. Gdy program zakończy używanie katalogu, powinien go zamknąć. Można to osiągnąć używając funkcji `closedir`.

Użycie

```
#include <dirent.h>
int closedir(DIR *dirptr);
```

Funkcja `closedir` zamknie strumień katalogu, wskazywany przez argument `dirptr`. Zwykle tym argumentem jest wartość zwracana przez wywołanie funkcji `opendir`, jak pokazuje następujący fragment kodu:

```
#include <stdlib.h>
#include <dirent.h>

main()
{
    DIR *dp;

    if((dp = opendir("/tmp/dir1")) == NULL)
    {
        fprintf(stderr, "Error on opening directory /tmp/dir1\n");
        exit(1);
    }

    /* kod przetwarzający katalog */
    .

    closedir(dp);
}
```

4.4.3 Czytanie katalogów: `readdir` i `rewinddir`

Gdy katalog zostanie już utworzony przez `opendir`, każda pozycja katalogu może być wczytyana do struktury `dirent`.

Użycie

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirptr);
```

Funkcji `readdir` należy przekazać prawidłowy wskaźnik strumienia katalogu, zwykle zwrócony przez wywołanie funkcji `opendir`. Przy pierwszym wywołaniu `readdir` do struktury `dirent` zostanie wczytana pierwsza pozycja katalogu. Po zakończeniu wskaźnik katalogu zostanie przesunięty na następną pozycję w katalogu.

Gdy po kolejnych wywołaniach `readdir` zostanie osiągnięty koniec katalogu, zwrócony będzie wskaźnik pusty. Jeśli w dowolnym momencie program zechce ponownie czytać od początku katalogu, można wywołać funkcję `rewinddir`, która jest zdefiniowana następująco:

Użycie

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dirptr);
```

Następujące po `rewinddir` ponowne wywołanie `readdir` zwróci pierwszą pozycję w katalogu wskazywanym przez `dirptr`.

W następującym przykładzie funkcja `my_double_ls` pokaże dwukrotnie listę nazw wszystkich plików w danym katalogu. Pobiera ona jako parametr nazwę katalogu i zwraca -1 w przypadku błędu.

```
#include <dirent.h>

int my_double_ls(const char *name)
{
    struct dirent *d;
    DIR *dp;

    /* otwiera katalog i sprawdza błędy */
    if((dp=opendir(name)) == NULL)
        return (-1);

    /* krały w pętli po katalogu,
     * drukując nazwy pozycji katalogu,
     * dopóki ważny jest numer i-węzła
     */
    while(d = readdir(dp))
    {
        if(d->d_ino != 0)
            printf("%s\n", d->d_name);
    }

    /* teraz powraca do początku katalogu ... */
    rewinddir(dp);

    /* ... i drukuje katalog ponownie */
    while(d = readdir(dp))
    {
        if(d->d_ino != 0)
```

```

        printf("%s\n", d->d_name);
    }

closedir(dp);
return (0);
}

```

Kolejność, w jakiej funkcja `my_double_ls` wypisze nazwy plików, odpowiada kolejności, w jakiej pliki zostały umieszczone w katalogu. Jeśli `my_double_ls` jest wykonywana w katalogu zawierającym trzy pliki abc, bookmark i fred, wynik może wyglądać następująco:

```

..
fred
bookmark
abc
.

..
fred
bookmark
abc

```

Przykład drugi: find_entry

Następna przykładowa procedura, `find_entry`, przegląda katalog w poszukiwaniu kolejnego wystąpienia pliku (lub podkatalogu) kończącego się określonym przyrostkiem. Funkcja pobiera trzy argumenty: nazwę przeszukiwanego katalogu, ciąg znaków przedrostka i znacznik określający, czy przeszukiwanie ma być kontynuowane od ostatnio znalezionej pozycji. Jeśli została znaleziona odpowiednia nazwa, będzie zwrócony wskaźnik do tej nazwy, a w przeciwnym wypadku – wskaźnik pusty.

`find_entry` używa procedury dopasowania napisów o nazwie `match`, aby sprawdzić, czy dana nazwa pliku kończy się odpowiednim przyrostkiem. `match` używa dwóch standardowych procedur z biblioteki C, obecnych we wszystkich systemach Unix: `strlen`, zwracającej długość napisu w znakach i `strcmp`, która porównuje dwa napisy, zwracając zero, gdy są identyczne.

```

#include <stdio.h>      /* dla NULL */
#include <dirent.h>
#include <string.h>      /* dla funkcji napisowych */

int match(const char *, const char *);

char *find_entry(char *dirname, char *suffix, int cont)
{
    static DIR *dp=NULL;
    struct dirent *d;

    if(dp == NULL || cont == 0)
    {
        if(dp != NULL)

```

```

            closedir(dp);
            if((dp = opendir(dirname)) == NULL)
                return NULL;
        }

        while(d=readdir(dp))
        {
            if(d->d_ino == 0)
                continue;
            if(match(d->d_name, suffix))
                return (d->d_name);
        }
        closedir(dp);
        dp = NULL;
        return (NULL);
    }

    int match(const char *s1, const char *s2)
    {
        int diff = strlen(s1) - strlen(s2);

        if(strlen(s1) > strlen(s2))
            return(strcmp(&s1[diff], s2) == 0);
        else
            return (0);
    }
}

Ćwiczenie 4.1 Zmodyfikuj funkcję my_double_ls z wcześniejszego ćwiczenia, aby pobierała drugi parametr – liczbę całkowitą o nazwie skip. Gdy skip jest ustaliona na 0, my_double_ls powinna działać jak poprzednio. Gdy skip ma wartość 1, my_double_ls powinna pominąć nazwy, które zaczynają się od kropki.

Ćwiczenie 4.2 W poprzednim rozdziale wprowadziliśmy funkcje systemowe stat i fstat jako sposób uzyskiwania informacji o pliku. Struktura stat, zwracana przez stat i fstat, zawiera pole o nazwie st_mode, określające tryb pliku. Składa się ono z praw dostępu do pliku sumowanych bitowo (OR-owanych) ze stałą, która określa, czy dana struktura reprezentuje zwykły plik, katalog, plik specjalny czy mechanizm komunikacji międzyprocesowej jak nazwany potok. Najlepszym sposobem sprawdzenia, czy dany plik jest katalogiem, jest użycie makroinstrukcji S_ISDIR:
```

```

/* buf pochodzi z wywołania stat */
if(S_ISDIR(buf.st_mode))
    printf("It's a directory\n");
else
    printf("It's not\n");

```

Zmodyfikuj procedurę `my_double_ls` tak, aby wywoływała `stat` dla każdego znalezionego pliku i pokazywała gwiazdkę po każdej nazwie, która odnosi się do katalogu.

4.4.4 Bieżący katalog roboczy

Jak już wiemy z podrozdziału 4.2, każdy zarejestrowany użytkownik pracuje we własnym bieżącym katalogu roboczym. W rzeczywistości każdy proces Unixa, to jest każdy wykonywany program, ma swój własny bieżący katalog roboczy. Jest on używany jako punkt startowy wszystkich względnych nazw ścieżek, przeszukiwanych przy wywoływaniu open i podobnych. Bieżący katalog roboczy użytkownika jest w istocie bieżącym katalogiem roboczym, związanym z procesem powłoki, która interpretuje polecenia użytkownika.

Początkowo bieżący katalog roboczy procesu jest ustawiany na bieżący katalog roboczy procesu, który go uruchomil (zazwyczaj powłoki). Jednak proces może zmienić swój bieżący katalog roboczy za pomocą funkcji systemowej chdir.

4.4.5 Zmiana katalogów za pomocą chdir

Użycie

```
#include <unistd.h>
int chdir(const char *path);
```

Funkcja systemowa chdir powoduje, że path staje się nowym bieżącym katalogiem roboczym wywolującego procesu. Ważne jest, że zmiana ta oddziałuje jedynie na proces, który wywołał chdir. W szczególności program, który zmienia katalog, nie przeszkadza powłoce, która ten program uruchomila. Tak więc, gdy program się zakończy, użytkownik znajdzie powłokę w miejscu, gdzie rozpoczął, bez względu na wewnętrzki programu.

Wywołanie funkcji chdir nie powiedzie się, zwracając przy tym wartość -1, jeśli path nie definiuje poprawnego katalogu albo jeśli prawo wykonania nie istnieje w każdym katalogu wzduż ścieżki dla procesu wywolującego.

Funkcja chdir przydaje się, gdy program potrzebuje dostępu do wielu plików w danym katalogu. Zmiana katalogu i używanie nazw plików względnych wobec tego katalogu jest bardziej efektywne niż używanie pełnych nazw plików. Dzieje się tak dlatego, że system musi znaleźć każdy katalog w ścieżce, zanim zlokalizuje właściwą nazwę pliku, więc zmniejszenie liczby elementów ścieżki oszczędza czas. Na przykład, zamiast używać takiego fragmentu programu:

```
fd1 = open("/usr/ben/abc", O_RDONLY);
fd2 = open("/usr/ben/xyz", O_RDWR);
```

programista może użyć następującego:

```
chdir("/usr/ben");
fd1 = open("abc", O_RDONLY);
fd2 = open("xyz", O_RDWR);
```

4.4.6 Znajdowanie nazwy bieżącego katalogu roboczego

XSI definiuje funkcję (nie funkcję systemową) o nazwie getcwd, która zwraca nazwę bieżącego katalogu roboczego.

Użycie

```
#include <unistd.h>
char *getcwd(char *name, size_t size);
```

Funkcja getcwd zwraca wskaźnik do nazwy ścieżki bieżącego katalogu. Powinieneś pamiętać, że wielkość argumentu size powinna być przynajmniej o jeden większa od długości nazwy, która ma być zwrócona. W przypadku sukcesu, nazwa aktualnego katalogu jest kopowana do tablicy, wskazywanej przez name. Wywołanie nie powiedzie się i zwróci pusty wskaźnik, jeśli size będzie równe 0. W niektórych implementacjach, jeśli name jest wskaźnikiem pustym, getcwd zaalokuje size bajtów pamięci dynamicznej; ponieważ jednak ta implementacja jest zależna od systemu, wywoływanie getcwd z pustym wskaźnikiem nie zaleca się.

Przykład: my_pwd

Ten krótki program naśladuje polecenie pwd.

```
/* my_pwd -- drukuje katalog roboczy */
```

```
#include <stdio.h>
#include <unistd.h>
#define VERYBIG 200

void my_pwd(void);

main()
{
    my_pwd();
}

void my_pwd(void)
{
    char dirname[VERYBIG];

    if(getcwd(dirname, VERYBIG) == NULL)
        perror("getcwd error");
    else
        printf("%s\n", dirname);
}
```

4.4.7 Przechodzenie drzewa katalogów

Czasem konieczne jest wykonanie operacji na hierarchii katalogów, zaczynając od pewnego katalogu początkowego i przechodząc w dół przez wszystkie pliki i podkatalogi. W tym celu Unix zapewnia procedurę o nazwie fts, która wykonuje przejście przez drzewo katalogów, poczynając od dowolnego katalogu i wykonując zdefiniowaną przez użytkownika procedurę dla każdej znalezionej pozycji katalogu.

Użycie

```
#include <ftw.h>
int ftw(const char *path, int (*func)(), int depth);
```

Pierwszy parametr, `path`, definiuje nazwę ścieżki katalogu, w którym powinno rozpoczęć się rekurencyjne przechodzenie drzewa. Parametr `depth` jest dość dziwnym modelem, który kontroluje liczbę różnych deskryptorów plików, używanych przez `ftw`. Im większa jest wartość `depth`, tym mniej katalogów musi być powtarzanie otwieranych, co zwiększa szybkość tego wywołania. Chociaż na każdym poziomie drzewa może być używany tylko jeden deskryptor, musisz dopilnować, żeby wartość `depth` nie była większa niż liczba dostępnych deskryptorów. Do pobrania maksymalnej liczby deskryptorów, które proces może zaalokować, można użyć funkcji systemowej `getrlimit`, omawianej w rozdziale 12.

Drugi parametr, `func`, to zdefiniowana przez użytkownika funkcja, która zostanie wywołana dla każdego pliku lub katalogu, znalezionego w hierarchii, poczynając od `path`. Jak widzieliśmy w opisie użycia, `func` jest przekazywana do procedury `ftw` jako wskaźnik do funkcji. Musi więc być zadeklarowana przed wywołaniem `ftw`. Za każdym razem `func` będzie wywołana z trzema argumentami: zakończonym zerem napisem przechowującym nazwę obiektu, wskaźnikiem do struktury `stat` zawierającej informacje o obiekcie oraz kodem będącym liczbą całkowitą. Zatem `func` powinna wyglądać następująco:

```
int func(const char *name, const struct stat *sptr, int type)
{
    /* ciało funkcji */
}
```

Argument liczbowy (w naszym przypadku `type`) zawiera jedną z kilku możliwych wartości (zdefiniowanych w pliku nagłówkowym `<ftw.h>`), która opisuje natopkany obiekt. Te wartości to:

- `FTW_F` Obiekt jest plikiem.
- `FTW_D` Obiekt jest katalogiem.
- `FTW_DNR` Obiekt jest katalogiem, który nie może być czytany.
- `FTW_SL` Obiekt jest łączem symbolicznym.
- `FTW_NS` Obiekt nie jest łączem symbolicznym i wywołanie `stat` nie powiedzie się.

Jeśli obiekt jest katalogiem, który nie może być czytany (`FTW_DNR`), to jego podkatalogi nie będą przetwarzane. Jeśli wywołanie funkcji `stat` nie może być wykonane pomyślnie (`FTW_NS`), to struktura `stat` będzie zawierała niezdefiniowane wartości.

Przechodzenie drzewa jest kontynuowane do osiągnięcia dołu drzewa lub do wystąpienia błędu w `ftw`. Przechodzenie może także zostać przerwane, jeśli zdefiniowana przez użytkownika funkcja zwróci wartość niezerową. W tym miejscu procedura `ftw` przerwie działanie i zwróci wartość, którą uzyskała od funkcji użytkownika. Błędy w `ftw` powodują zwrócenie wartości -1 i ustawienie typu błędu w `errno`.

Następny przykład używa `ftw` do przejścia w dół hierarchii katalogu, drukując nazwy natopkanych po drodze plików, razem z ich prawami dostępu, i wskazując, które pliki są katalogami (lub łączami symbolicznymi) przez dodanie gwiazdki do nazwy.

Przyjrzyjmy się najpierw funkcji `list`, która będzie przekazywana jako argument do `ftw`.

```
#include <sys/stat.h>
#include <ftw.h>

int list(const char *name, const struct stat *status, int type)
{
    /* powrót, jeśli wywołanie stat nie powiedzie się */
    if(type == FTW_NS)
        return 0;

    /*
     * w innym wypadku wypisz nazwę obiektu,
     * pozwolenia oraz przyrostek "*",
     * jeżeli obiekt jest katalogiem lub łączem symbolicznym
     */
    if(type == FTW_F)
        printf("%-30s\t%03o\n", name, status->st_mode&0777);
    else
        printf("%-30s*\t%03o\n", name, status->st_mode&0777);

    return 0;
}
```

Nastepnym zadaniem jest napisanie programu głównego, który akceptuje nazwę ścieżki jako argument i używa jej jako punktu początkowego do przechodzenia drzewa katalogu. Jeżeli argumenty nie są dostarczone, przechodzenie rozpocznie się w bieżącym katalogu roboczym.

```
main(int argc, char **argv)
{
    int list(const char *, const struct stat *, int)

    if(argc == 1)
        ftw(".", list, 1);
    else
        ftw(argv[1], list, 1);
    exit(0);
}
```

W przypadku prostej hierarchii katalogów wyjście naszego programu `list` może wyglądać następująco:

```
$ list
./list      * 0755
./file1     0644
./subdir    * 0777
./subdir/another 0644
./subdir/subdir2   * 0755
./subdir/yetanother 0644
```

Zwróć uwagę, w jakiej kolejności są przeszukiwane katalogi.

4.5 Systemy plików Uniksa

Widzieliśmy, że pliki mogą być zorganizowane w różne katalogi, które z kolei stanowią część całkowitej hierarchicznej struktury drzewa. Same katalogi mogą być grupowane w **system plików** (ang. *filesystem*). Zasadniczo systemy plików stanowią obiekt zainteresowania tylko administratora systemu Unix. Pozwalają one na przystosowanie struktury katalogu do szeregu różnych fizycznych dysków lub partycji dyskowych, zachowując dla użytkownika jednolity wygląd struktury.

Każdy system plików zaczyna się przy węźle katalogu w hierarchii drzewa. Ta własność pozwala administratorom systemu dzielić hierarchię plików Uniksa i alokować jej części w określonych obszarach dysku lub wręcz dzielić całą strukturę plików na kilka fizycznych urządzeń dyskowych.

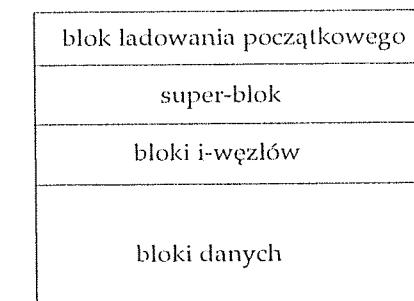
Systemy plików są także nazywane **odłączalnymi wolumenami** (ang. *demountable volumes*), ponieważ możliwe jest dynamiczne wprowadzanie całych podsekcji hierarchii gdziekolwiek w strukturze drzewa. Co więcej, możliwe jest też dynamiczne odłączanie lub dołączanie całych systemów plików w hierarchii tak, że będą czasowo niedostępne dla użytkowników. W każdej chwili kilka systemów plików może być bezpośrednio dołączonych, ale nie wszystkie będą widoczne jako część hierarchii struktury drzewa.

Informacja zawarta w systemie plików znajduje się na partycji dysku, która jest identyfikowana przez jednostkę nazywaną **plikiem urządzenia** (ang. *device file*) lub **plikiem specjalnym** Uniksa i której idea jest szczegółowo omówiona poniżej. Na razie zwróć uwagę, że systemy plików są jednoznacznie identyfikowane przez nazwy plików Uniksa.

Rzeczywiste niskopoziomowe rozmieszczenie danych zawarte w systemie plików jest całkowicie różne od wysokopoziomowego widoku hierarchii katalogów przedstawianego użytkownikowi. W dodatku, to rozmieszczenie nie jest częścią *XSI* i istnieje wiele jego różnych form. Jądro może jednocześnie obsługiwać systemy plików z różnymi rozmieszczeniami. Poniżej opisujemy tradycyjne rozmieszczenie.

Tradycyjny system plików jest podzielony na pewną liczbę logicznych bloków. Każdy taki system plików zawiera cztery różne części: obszar ładowania początkowego (ang. *bootstrap*), super-blok systemu plików, pewną liczbę bloków zarezerwowanych dla struktur i-węzłów systemu plików oraz obszar zarezerwowany dla bloków danych, które tworzą pliki w tym konkretnym systemie plików. Rozmieszczenie to jest pokazane na rysunku 4.6. Pierwszy z bloków (logicznie blok 0 w systemie plików, ale fizycznie gdzieś na początku partycji dyskowej) jest zarezerwowany do użycia jako blok ładowania początkowego. To znaczy, że może on zawierać specyficzny dla sprzętu program ładowający, używany do ładowania Uniksa w czasie startu systemu.

Logiczny blok 1 jest nazywany super-blokiem systemu plików. Zawiera całą życiową informację o systemie plików, na przykład całkowitą wielkość systemu plików (r bloków na rysunku), liczbę bloków zarezerwowanych dla i-węzłów ($n - 2$), i czas ostatniej aktualizacji systemu plików. W super-bloku znajdują się też dwie listy. Pierwsza zawiera część łańcucha numerów wolnych bloków danych liczb, a druga – część łańcucha wolnych numerów i-węzłów w celu szybkiego dostępu przy alokacji nowych bloków dysku dla danych albo tworzenia nowej pozycji katalogu. Super-blok dla dołączonego systemu plików trzymany jest w pamięci, aby uzyskać szybki dostęp do list wolnych bloków i wolnych i-węzłów. Kiedy listy znajdujące się w pamięci zostaną wyczerpane, są uzupełniane z dysku.



Rysunek 4.6 Tradycyjne rozmieszczenie systemu plików

Wielkość struktury i-węzła zależy od systemu plików; na przykład w kilku systemach ma ona wielkość 64 bajtów, a w innych może mieć 128 bajtów. I-węzły są numerowane kolejno, począwszy od 1, a do określenia położenia struktury i-węzła na podstawie jego numeru pobranego z pozycji katalogu używany jest prosty algorytm.

Systemy plików są tworzone przy użyciu programu `mksfs` i gdy ten program jest wykonywany, musi być określona wielkość obszarów i-węzłów i obszarów danych dla systemu plików. W tradycyjnych systemach plików te wielkości nie mogły być zmieniane dynamicznie i możliwe były dwa sposoby wyczerpania przestrzeni systemu plików. Po pierwsze, wszystkie bloki danych mogły być zużyte (nawet jeśli były jeszcze dostępne wolne numery i-węzłów). Po drugie, mogły być zużyte wszystkie numery i-węzłów (przez tworzenie dużej liczby małych plików) i w kon-

sekwencji nie można było utworzyć żadnego nowego pliku w systemie plików, chociaż były wolne bloki danych. Jednak obecnie nowsze systemy plików mogą mieć zmienianą wielkość, a i-węzły są alokowane dynamicznie.

Powyższy wywód dowodzi, że numery i-węzłów są unikatowe tylko w obrębie systemu plików i dlatego niemożliwe jest łącze do pliku przez systemy plików, jeśli nie są używane łącza symboliczne.

4.5.1 Buforowanie w pamięci podręcznej: sync i fsync

W tradycyjnej strukturze systemu plików w celu zapewnienia efektywności kopie super-bloków dołączonych systemów plików są trzymane w pamięci głównej komputera uniksowego. Aktualizacja tych super-bloków może wtedy następować bardzo szybko, bez potrzeby bezpośredniego dostępu do dysku. Podobnie, wszystkie transfery z pamięci do dysku, to jest zapisywania, są zazwyczaj buforowane w przestrzeni danych systemu operacyjnego, zamiast wpisywania ich od razu na dysk. Także odczyty są buforowane w pamięci podręcznej. Dlatego w każdej chwili, dane będące na dysku mogą być nieaktualne w stosunku do danych buforowanych w pamięci. Unix posiada dwie funkcje, pozwalające procesowi na upewnienie się, że wszystko jest aktualne. Funkcja sync jest używana do zapisu na dysk wszystkich buforów zawierających informacje o systemach plików, a fsync – do zapisania danych i atrybutów związanych z określonym plikiem.

Użycie

```
#include <unistd.h>
void sync(void);
int fsync(int filedes);
```

Ważną różnicą pomiędzy tymi dwoma wywołaniami jest to, że fsync nie zwróci sterowania, dopóki wszystkie dane nie zostaną zapisane na dysk, a wywołanie funkcji sync może zwrócić, gdy zapisanie danych zostanie zaplanowane, ale niekoniecznie ukończone (a w pewnych specyficznych implementacjach, sync może być niepotrzebne i nie powodować żadnego efektu).

Jak widzimy, funkcja sync nie zwraca wartości. Wywołanie fsync zwróci 0 w przypadku sukcesu, a -1 przy błędzie. Funkcja fsync może zanieść na przykład wtedy, gdy filedes nie jest prawidłowym deskryptorem pliku.

Aby zapewnić dużą aktualność systemów plików, system Unix ciągle wykonuje fragment kodu, który wielokrotnie wywołuje sync, czekając określony czas pomiędzy wywołaniami. Zazwyczaj ten okres wynosi 30 sekund, chociaż ów czas jest parametrem, który może zostać skonfigurowany przez administratora systemu.

4.6 Pliki urządzeń Uniksa

Urządzenia zewnętrzne, dołączone do systemów Unix (dyski, terminale, drukarki, jednostki taśmowe i inne), są dostępne przez nazwy plików w systemie plików. Pliki te zwane są plikami urządzeń. Parytce dysku odpowiadające systemom plików są jednymi z obiektów, reprezentowanymi przez te specjalne pliki.

W odróżnieniu od zwykłych plików dyskowych, odczyty i zapisy do tych plików urządzeń powodują bezpośrednią transmisję danych pomiędzy systemem i odpowiednim urządzeniem zewnętrznym.

Zazwyczaj te specjalne pliki są przechowywane w katalogu o nazwie /dev. Na przykład:

```
/dev/tty00
/dev/console
/dev/pts/as      (pseudoterminal, dostępny z sieci)
```

mogą być nazwami trzech systemowych portów terminali, a:

```
/dev/lp
/dev/rmt0
/dev/rmt/0cbn
```

mogą odnosić się do drukarki i dwóch napędów taśmy magnetycznej.

Nazwy partycji dyskowych mogą się znacznie różnić. Przykładowe możliwości to:

```
/dev/dsk/c0b0t0d0s3
/dev/dsk/hd0d
```

Pliki urządzeń mogą być używane na poziomie powłoki lub w programach dokładnie tak, jak zwykłe pliki. Na przykład polecenia:

```
$ cat fred > /dev/lp
$ cat fred > /dev/rmt0
```

spowodują odpowiednio wydrukowanie pliku fred na drukarce wierszowej i zapisanie go na taśmie magnetycznej (z uwzględnieniem praw dostępu). Oczywiście szaleństwem byłoby manipulowanie w ten sposób partycjami dyskowymi, zawierającymi systemy plików. Wiele cennych informacji może być przypadkowo zniszczonych przez nieprzymyślany rozkaz. W dodatku, gdyby pozwolenia dla takiego pliku urządzenia były zbyt pobłażliwe, zdolni użytkownicy mogliby pominać prawa dostępu dla plików trzymanych w systemie plików. Aby stało się to niewykonalne, administratorzy systemów powinni ustanowić odpowiednie prawa dostępu dla plików urządzeń odpowiadających partycjom dyskowym.

Z programu w stosunku do plików urządzeń mogą być używane funkcje open, close, read i write. Na przykład:

```
#include <fcntl.h>

main()
{
    int i, fd;
    fd = open("/dev/tty00", O_WRONLY);
    for(i=0; i<100; i++)
```

```

    write(fd, "x", 1);

close(fd);
}

```

spowoduje wypisanie sto razy x na terminalu `tty00`. Obsługa terminala jest oczywiście ważnym tematem i omówimy go bardziej szczegółowo w rozdziale 9.

4.6.1 Pliki urządzeń blokowych i znakowych

Pliki urządzeń Uniksa są podzielone na dwie kategorie: **urządzenia blokowe** i **urządzenia znakowe**.

1. *Pliki urządzeń blokowych* zawierają urządzenia takie jak dyski lub taśmy magnetyczne. Transfer danych pomiędzy tymi urządzeniami a jądrem odbywa się za pomocą standardowej wielkości bloków. Wszystkie urządzenia blokowe umożliwiają bezpośredni dostęp. Wewnątrz jądra, dostęp do tych urządzeń jest kontrolowany przez uporządkowany zestaw struktur danych jądra i procedur. Ten wspólny interfejs urządzeń blokowych oznacza, że najczęściej sterowniki urządzeń blokowych są bardzo podobne, różniąc się tylko niskopoziomowym sterowaniem urządzeń.
2. *Pliki urządzeń znakowych* zawierają takie urządzenia, jak linie terminali, linie modemowe i urządzenia drukujące, nie posiadające strukturalnego mechanizmu transferu. Bezpośredni dostęp może, ale nie musi być obsługiwany. Transfer danych nie jest przeprowadzany za pomocą bloków stałej wielkości, ale strumieniami bajtów dowolnej długości.

Należy pamiętać, że systemy plików mogą istnieć tylko na urządzeniach blokowych, dlatego te urządzenia mają powiązane znakowe urządzenia szybkiego dostępu, znane często jako urządzenia pierwotne (ang. *raw*). Narzędzia takie jak `mkfs` i `fsck` używają surowego interfejsu.

Unix używa dwóch tablic konfiguracyjnych systemu operacyjnego, nazywanych tablicą **przełączników urządzeń blokowych** (ang. *block device switch*) i tablicą **przełączników urządzeń znakowych** (ang. *character device switch*), do powiązania urządzenia zewnętrznego ze specyficzny dla tego urządzenia kodem, który nim steruje. Te tablice i kod specyficzny dla urządzeń zawarte są w samym jądrze. Obie tablice są indeksowane za pomocą wartości zwanej **głównym numerem urządzenia** (ang. *major device number*), która jest przechowywana w i-węźle pliku urządzenia. Kolejność transmisji danych do lub z urządzenia zewnętrznego jest następująca:

1. Funkcje systemowe `read` i `write` używają i-węzła pliku urządzenia w zwykły sposób.
2. System sprawdza znacznik w strukturze i-węzła, aby wiedzieć, czy to urządzenie jest urządzeniem **blokowym**, czy **znakowym**. Pobrany zostaje także numer główny.
3. Numeru głównego używa się do indeksowania odpowiedniej tablicy konfiguracji urządzenia. W celu wykonania transferu danych wywoływana jest specyficzna dla danego urządzenia procedura sterownika.

Dzięki temu dostęp do urządzeń zewnętrznych jest całkowicie zgodny z dostępem do zwykłych plików dyskowych.

Oprócz głównego numeru urządzenia, w i-węźle przechowywany jest także **poboczny numer urządzenia** (ang. *minor device number*), przekazywany do procedur sterownika urządzenia w celu identyfikacji udostępnianego portu urządzenia (jeśli urządzenie posiada więcej niż jeden port zewnętrzny). Na przykład w 8-liniowej karcie terminali, każda linia terminala będzie współdzieliła ten sam numer główny (i konsekwentnie ten sam zestaw procedur sterownika urządzenia), ale każda będzie miała własny niepowtarzalny numer poboczny z przedziału od 0 do 7, który będzie identyfikował używaną linię.

4.6.2 Powtórnie struktura stat

Struktura `stat`, przedstawiana w rozdziale 3, przechowuje informacje o pliku urządzenia w dwóch polach:

`st_mode` W przypadku pliku urządzenia, zawiera ono uprawnienia dla pliku dodane do ósemkowej wartości 060000 dla urządzeń blokowych, a 020000 dla urządzeń znakowych. W `<stat.h>` zdefiniowano stałe symboliczne o nazwach `S_IFBLK` i `S_IFCHR`, które mogą być używane zamiast tych liczb.

`st_rdev` Zawiera ono główny i poboczny numer urządzenia.

Ta informacja może być wyświetlona poprzez użycie polecenia `ls` z opcją `-l`. Na przykład:

```
$ ls -l /dev/tty3
crw--w--w- 1 ben other 8,3 Sep 13 10:19 /dev/tty3
```

Zauważ literę `c` w pierwszej kolumnie wyjścia, która oznacza, że `/dev/tty3` jest plikiem urządzenia znakowego. Wartości całkowite 8 i 3 reprezentują odpowiednio główny i poboczny numer urządzenia.

W programie, pole `st_mode` może być sprawdzone za pomocą techniki przedstawionej w ćwiczeniu 4.2:

```
if(S_ISCHR(buf.st_mode))
    printf("It's a character device\n");
else
    printf("It's not\n");
```

Pamiętaj, że `S_ISCHR` jest makroinstrukcją, zdefiniowaną w `<stat.h>`.

4.6.3 Informacja o systemie plików

Dla urządzeń, które zawierają systemy plików, dwie funkcje, `statvfs` i `fstatvfs`, mogą być używane do pobrania podstawowych informacji o systemie plików, takich jak całkowita liczba wolnych bloków dysku lub liczba wolnych i-węzłów.

Użycie

```
#include <sys/statvfs.h>
int statvfs(const char *path, struct statvfs *buf);
int fstatvfs(int fd, struct statvfs *buf);
```

Obie funkcje zwracają informacje o systemie plików, który zawiera plik określony przez parametr path w statvfs lub otwarty plik, określony przez fd w fstatvfs. Parametr buf wskazuje na typ struktury statvfs, który jest zdefiniowany w pliku nagłówkowym <sys/statvfs.h>. Struktura statvfs zawiera co najmniej następujące składowe:

unsigned long f_bsize	Wielkość bloku systemu plików, używana przez system do zwiększenia efektywności. Na przykład składowa f_frsize może być ustaliona na 1K, ale f_bsize – na 8K, bo system będzie wtedy bardziej efektywnie wykonywać operacje wejścia-wyjścia.
unsigned long f_frsize	Podstawowa wielkość bloku systemu plików (określona przy tworzeniu systemu plików).
unsigned long f_blocks	Calkowita liczba bloków w systemie plików w jednostkach f_frsize.
unsigned long f_bfree	Calkowita liczba wolnych bloków.
unsigned long f_bavail	Liczba wolnych bloków dostępnych dla nieuprzywilejowanych procesów.
unsigned long f_files	Calkowita liczba numerów i-węzłów.
unsigned long f_ffree	Calkowita liczba wolnych numerów i-węzłów.
unsigned long f_favail	Liczba wolnych numerów i-węzłów dostępnych dla nieuprzywilejowanych procesów.
unsigned long f_fsid	id systemu plików.
unsigned long f_flag	Maska bitowa wartości znacznika.
unsigned long f_namemax	Maksymalna długość pliku.

Następny przykład jest odpowiednikiem standardowej komendy df. Używa on statvfs do wydruku liczby wolnych bloków i wolnych i-węzłów w systemie plików.

```
/* fsys -- drukuje informacje o systemie plików. */
/* nazwa systemu plików jest przekazywana jako argument */
#include <sys/statvfs.h>
```

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

main(int argc, char **argv)
{
    struct statvfs buf;

    if(argc != 2)
    {
        fprintf(stderr, "usage: fsys filename\n");
        exit(1);
    }

    if(statvfs(argv[1], &buf) != 0)
    {
        fprintf(stderr, "statvfs error\n");
        exit(2);
    }
    printf("%s:\tfree blocks %d\tfree inodes %d\n",
           argv[1], buf.f_bfree, buf.f_ffree);
    exit(0);
}
```

4.6.4 Ograniczenia plików i katalogów: pathconf i fpathconf

POSIX i inne próby standaryzacji bardziej sformalizowały pewne ograniczenia systemowe. Ponieważ system może obsługiwać wiele typów systemów plików, niektóre ograniczenia mogą się różnić. Dwie procedury, pathconf i fpathconf mogą być używane do poznania ograniczeń twojego systemu na bazie plików lub katalogów.

Użycie

```
#include <unistd.h>
long int pathconf(const char *pathname, int name);
long int fpathconf(int filedes, int name);
```

Obie procedury działają w ten sam sposób i zwracają tę samą wartość dla żądanego limitu lub zmiennej. Różni je pierwszy argument: pathconf powinien dostać prawidłową nazwę ścieżki do pliku lub katalogu, a fpathconf deskryptor otwartego pliku. Drugim parametrem jest symboliczna nazwa zmiennej dotyczącej danego pliku lub katalogu, pobrana z <unistd.h>.

Następująca procedura, lookup, może być użyta w pliku do wyświetlenia ograniczeń systemowych dla danego pliku. W tym przykładzie lookup wyświetla najbardziej interesujące z tych wartości dla katalogu /tmp.

```
/* lookup - wyświetla ustawienia ograniczeń pliku */
#include <unistd.h>
#include <stdio.h>
```

```

typedef struct{
    int val;
    char *name;
} Table;

main()
{
    Table *tb;
    static Table options[] = {
        { _PC_LINK_MAX, "Maximum number of links" },
        { _PC_NAME_MAX, "Maximum length of a filename" },
        { _PC_PATH_MAX, "Maximum length of a pathname" },
        {-1, NULL}
    };

    for(tb=options; tb->name != NULL; tb++)
        printf("%-28.28s\t%ld\n", tb->name,
               pathconf("/tmp", tb->val));
}

```

Gdy uruchomiono ten program na pewnym systemie, wyświetlił następujące wyniki:

```

Maximum number of links      32767
Maximum length of a filename 256
Maximum length of a pathname 1024

```

/tmp jest katalogiem. Maksymalna liczba łączy odnosi się do samego katalogu. Jednak maksymalna długość nazwy pliku dotyczy plików w katalogu. Generalnie rzecz biorąc, ograniczenia systemowe są udokumentowane w <limits.h>, a wartości mogą być pobrane dynamicznie, przez użycie podobnej procedury zwanej sysconf.

ROZDZIAŁ 5

Proces

- 5.1 Pojęcie procesu – raz jeszcze
- 5.2 Tworzenie procesów
- 5.3 Uruchamianie nowych programów za pomocą exec
- 5.4 Łączne użycie exec i fork
- 5.5 Dziedziczenie danych i deskryptorów plików
- 5.6 Kończenie procesów za pomocą funkcji systemowej exit
- 5.7 Synchronizacja procesów
- 5.8 Przedwczesne zakończenia i procesy zombie
- 5.9 Procesor poleceń: smallsh
- 5.10 Atrybuty procesu

5.1 Pojęcie procesu – raz jeszcze

Jak widzieliśmy w rozdziale 1, proces w pojęciu Uniksa jest po prostu egzemplarzem wykonującego się programu, co odpowiada pojęciu zadania (ang. *task*) w innych środowiskach. Każdy proces zawiera kod programu, wartości danych w postaci zmiennych programu i bardziej egzotyczne elementy, jak wartości przechowywane w rejestrach sprzętowych, stos programu itd.¹

Powłoka tworzy nowy proces za każdym razem, gdy uruchamia program w odpowiedzi na polecenie. Na przykład polecenie:

```
$ cat file1 file2
```

spowoduje utworzenie przez powłokę procesu wykonującego polecenie cat. Trochę bardziej skomplikowane polecenie:

```
$ ls | wc -l
```

1. Nie należy mylić tego z pojęciem wątku (ang. *thread*), gdzie wiele kopii kodu działa na jednym zestawie danych. Wątki są teraz dostępne w niektórych implementacjach Uniksa; opisano je w ostatnich rozszerzeniach POSIX i uaktualnieniach XSI. Nie będziemy jednak dalej opisywać złożoności modelu opartego na wątkach. Więcej informacji powinieneś znaleźć w podręczniku systemowym.

spowoduje jednoczesne utworzenie dwóch procesów do wykonywania poleceń ls i wc. (Dodatkowo wyjście programu ls wyświetlającego zawartość katalogu jest połączone za pomocą potoku (ang. *pipe*) z wejściem programu wc liczącego słowa). Ponieważ proces odpowiada wykonaniu programu, procesów nie należy mylić z programami, które one wykonują; wiele procesów może jednocześnie wykonywać ten sam program. Na przykład kilku użytkowników może równocześnie używać tego samego programu edytora, a każde wywołanie edytora jest liczone jako oddzielny proces.

Każdy proces Unixa może zacząć inne procesy. Daje to środowisku procesów Unixa strukturę hierarchiczną, naśladującą drzewo katalogów w systemie plików. Na wierzchołku drzewa procesów jest jeden proces sterujący, wykonanie niezwykle ważnego programu zwanego init, będącego przodkiem wszystkich procesów systemowych i procesów użytkowników.

Unix dostarcza programistom kilka funkcji systemowych do tworzenia i manipulacji procesami. Najważniejsze z nich (wyłączając funkcje komunikacji międzymiędzyprocesowej) to:

- fork** Używana do tworzenia nowych procesów poprzez powielenie procesu wywołującego. fork jest podstawową operacją tworzenia procesów.
- exec** Rodzina procedur bibliotecznych i jedna funkcja systemowa; każda z nich wykonuje tę samą podstawową funkcję: przekształcenie procesu wskutek podmiany jego przestrzeni pamięci przez nowy program. Różnice pomiędzy wywołaniami funkcji exec wynikają głównie z różnego sposobu budowy ich listy argumentów.
- wait** Ta funkcja zapewnia elementarną synchronizację procesów. Pozwala ona jednemu procesowi czekać, aż zakończy się inny, powiązany proces.
- exit** Używana do zakończenia procesu.

W dalszej części tego rozdziału omówimy ogólnie procesy Unixa, a szczegółowo te cztery ważne funkcje.

5.2 Tworzenie procesów

5.2.1 Funkcja systemowa fork

Podstawa tworzenia procesów to funkcja systemowa fork. Jest ona mechanizmem, który przekształca Unixa w środowisko wielozadaniowe.

Użycie

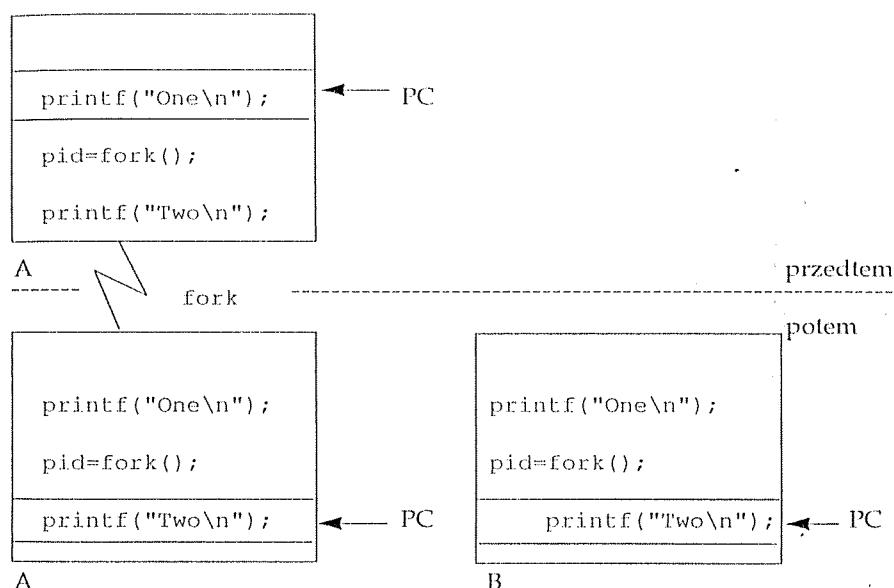
```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Zakończone sukcesem wywołanie fork powoduje stworzenie przez jądro nowego procesu, który jest (mniej lub bardziej) dokładną kopią procesu wywołującego. Innymi słowy, nowy proces uruchamia kopię tego samego programu, ze zmiennymi posiadającymi taką samą wartość jak te w procesie wywołującym, z jednym ważnym wyjątkiem, który omówimy za chwilę.

Nowo stworzony proces jest określany jako **proces potomny** (ang. *child process*), natomiast ten, który najpierw wywołał fork, jest nazywany **procesem rodzicielskim** (ang. *parent process*).

Po tym wywołaniu proces rodzicielski i jego nowo stworzony potomek wykonują się równocześnie; oba procesy zaczynają wykonanie od następnej instrukcji po wywołaniu fork.

Dla osób używających czysto sekwencyjnego środowiska programistycznego, idea fork może być na początku trudna do zrozumienia. Powinien to ułatwić rysunek 5.1. Koncentrujemy się tu na trzech liniach kodu, składających się z wywołania printf, następującego po nim wywołania fork i jeszcze jednego wywołania printf.



Rysunek 5.1 Wywołanie fork

Są dwie części tego rysunku, przedtem i potem. Część przedtem pokazuje stan rzeczy przed wywołaniem fork. Jest tu pojedynczy proces z etyktą A (używamy etykiety wyłącznie dla wygody; ona nie znaczy dla systemu). Strzałka z etykietą PC (co oznacza licznik programu (ang. *program counter*)) pokazuje aktualnie wykonywaną instrukcję. Ponieważ wskazuje ona pierwsze wywołanie printf, na standardowym wyjściu jest wyświetlany raczej trywialny komunikat One.

Część *potem* pokazuje położenie następujące bezpośrednio po wywołaniu `fork`. Teraz są tam dwa procesy *A* i *B*, działające razem. Proces *A* jest tym samym, co w części *przedtem* rysunku. *B* jest nowym procesem, zapoczątkowanym przez wywołanie `fork`. Z jednym większym wyjątkiem, wartości `pid`, jest on kopią *A* i wykonuje ten sam program co *A*, stąd podwojenie na rysunku trzech linii kodu źródłowego. Używając terminologii, którą wprowadziliśmy powyżej, powiemy, że *A* jest procesem rodzicielskim, podczas gdy *B* to jego potomek.

Dwie strzałki *PC* w tej części rysunku pokazują, że następna instrukcja wykonywana przez proces rodzicielski i proces potomny po wywołaniu `fork` to wywołanie `printf`. Innymi słowy, *A* i *B* znajdują się w tym samym miejscu w kodzie programu, chociaż proces *B* jest nowy dla systemu. Dlatego komunikat Two jest wyświetlany dwa razy.

Identyfikatory procesów

Jak widziales w opisie użycia na początku tego podrozdziału, funkcja `fork` jest wywoływana bez argumentów i zwraca `pid_t`. Typ specjalny `pid_t` zdefiniowany został w `<sys/types.h>` jako (najczęściej) liczba całkowita. Oto przykład wywołania:

```
#include <unistd.h> /* włącza definicję pid_t */

pid_t pid;
.

.

pid = fork();
```

Jest to wartość `pid`, która odróżnia proces potomny od rodzicielskiego. W procesie rodzicielskim, `pid` ustawiamy na niezerową, dodatnią liczbę. W procesie potomnym jest on ustawiany na zero. Ponieważ wartość zwracana przez oba procesy różni się, programista może określić różne działania dla tych procesów.

Liczba zwracana do procesu rodzicielskiego w `pid` została nazywana **identyfikatorem procesu** (ang. *process-id*) dla procesu potomnego. Jest to liczba, która identyfikuje nowy proces dla systemu, tak jak user-id identyfikuje użytkownika. Ponieważ wszystkie procesy powstają przez wywołanie funkcji `fork`, każdy proces Uniksa ma swój własny process-id, który jest zawsze unikatowy dla tego procesu. Dlatego process-id może być traktowany jako sygnatura identyfikująca swój proces.

Następujący krótki program obrazuje działanie `fork` i użycie process-id:

```
/* spaw -- pokazuje działanie fork */
#include <unistd.h>

main()
{
    pid_t pid; /* zawiera process-id rodzica */

    printf ("Just one process so far\n");
    printf ("Calling fork... \n");

    pid = fork(); /* utwórz nowy proces */
```

```
if(pid == 0)
    printf ("I'm the child\n");
else if(pid > 0)
    printf ("I'm the parent, child has pid %d\n", pid);
else
    printf ("Fork returned error code, no child\n");
```

Po wywołaniu `fork` znajdują się tu trzy rozgałęzienia instrukcji `if`. Pierwsze określa proces potomny, odpowiadający zerowej wartości zmiennej `pid`. Drugie działa w procesie rodzicielskim, odpowiadającym dodatniej wartości `pid`. Trzecie działa domyślnie, przy ujemnej wartości (w rzeczywistości -1) `pid`, która powstaje, kiedy funkcja `fork` zawiedzie przy tworzeniu procesu potomnego. Może to wskazywać, że proces wywołujący próbował przełamać jedno z dwóch ograniczeń; pierwsze jest systemowym ograniczeniem liczby procesów; drugie ogranicza liczbę procesów, które indywidualny użytkownik może uruchomić równocześnie. W obu okolicznościach zmienna `errno` zawiera kod błędu `EAGAIN`. Zauważ także, że ponieważ oba procesy generowane przez przykładowy program będą działać równocześnie i bez synchronizacji, nie ma żadnej gwarancji, że wyjście z procesu rodzicielskiego i procesu potomnego nie zostaną pomieszczone.

Zanim przejdziemy do dalszych zagadnień, warto zastanowić się, dlaczego funkcja `fork` jest w istocie przydatna, chociaż osamotniona może wyglądać na bezzużyteczną. Dzieje się tak głównie dla tego, że funkcja `fork` staje się wartościowa w połączeniu z innymi możliwościami Uniksa. Na przykład możemy mieć proces potomny i proces rodzicielski, które wykonują różne, ale powiązane zadania, współpracujące za sobą za pomocą jednego z mechanizmów komunikacji międzyprocesowej Uniksa, takich jak sygnały albo potoki (opisane dalej). Inną właściwością często używaną w połączeniu z `fork` jest funkcja systemowa `exec`, którą omówimy w następnym podrozdziale i która pozwala wykonywać inne programy.

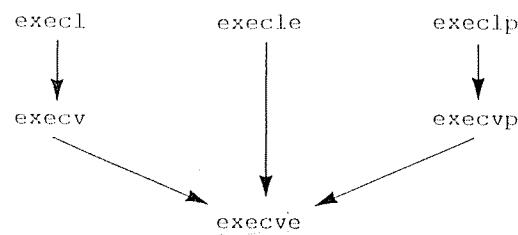
Ćwiczenie 5.1 Program może wywoływać `fork` kilka razy. Podobnie, każdy proces potomny może używać `fork`, aby utworzyć własne potomki. Aby to udowodnić, napisz program, który tworzy dwa podprocesy. Każdy z nich powinien następnie utworzyć jeden własny podproces. Po każdym wywołaniu `fork`, każdy proces rodzicielski powinien użyć `printf` do wyświetlenia process-id swojego potomka.

5.3 Uruchamianie nowych programów za pomocą exec

5.3.1 Rodzina exec

Gdyby `fork` stanowiło jedyną dostępną dla programisty podstawową operację tworzenia procesu, Unix byłby nieciekawy, ponieważ mogłyby być tworzone tylko kopie tego samego programu. Na szczęście do zapoczątkowania wykonywania nowego programu mogą być używani członkowie rodziny funkcji `exec`. Rysunek 5.2 pokazuje drzewo genealogiczne wywołania `exec`. Główną różnicą pomiędzy tymi funkcjami jest sposób przekazywania parametrów. Jak tego dowodzi rysunek,

wszystkie te funkcje ostatecznie wykonują wywołanie `execve` – rzeczywistej funkcji systemowej.



Rysunek 5.2 Drzewo rodziny funkcji `exec`

Następujący opis użycia pokazuje większość członków rodziny (niebawem powróćmy do `execle` i `execve`).

Użycie

```
#include <unistd.h>

/* rodzina funkcji execl musi mieć podane argumenty
   jako listę zakończoną wskaźnikiem NULL */
/* funkcja execl musi mieć podaną ważną nazwę ścieżki
   dla pliku wykonywalnego */
int execl(const char *path,
          const char *arg0, ..., const char *argn,
          (char *)0);

/* funkcja execlp potrzebuje tylko nazwy pliku wykonywalnego */
int execlp(const char *file,
            const char *arg0, ..., const char *argn,
            (char *)0);

/* rodzina funkcji execv musi mieć podaną tablicę argumentów */
/* funkcja execv musi mieć podaną ważną nazwę ścieżki
   do pliku wykonywalnego */
int execv(const char *path, char *const argv[]);

/* funkcja execvp potrzebuje tylko nazwy pliku wykonywalnego */
int execvp(const char *file, char *const argv[]);
```

Wszystkie warianty `exec` wykonują to samo zadanie: przekształcają proces wywołujący przez załadowanie nowego programu do jego przestrzeni pamięci. Jeśli wywołanie `exec` jest pomyślne, wywołujący program zostaje całkowicie przykryty przez nowy program, wykonywany następnie od samego początku. Wynik może być traktowany jako nowy proces, ale zachowuje ten sam process-id, co proces wywołujący.

Ważne jest podkreślenie, że `exec` nie tworzy nowego podprocesu działającego jednocześnie z procesem wywołującym. W zamian stary program jest wymazywany przez nowy. Tak więc, w odróżnieniu od `fork`, nie ma żadnego powrotu z pomyślnego wywołania funkcji `exec`.

Nieco upraszczając, omówimy dokładniej tylko jedną z funkcji `exec`, mianowicie `execl`.

Wszystkie parametry dla `execl` są wskaźnikami znakowymi. Pierwszy parametr, `path`, podaje nazwę pliku zawierającego program, który ma być wykonywany; przy funkcji `execl` musi to być ważna nazwa ścieżki, bezwzględna (pełna nazwa) albo względna. Sam plik musi też zawierać prawdziwy program albo skrypt powłoki z prawem wykonania. System sprawdza, czy plik zawiera program, przez przyjrzenie się jego dwóm pierwszym bajtom. Jeśli zawierają one specjalną wartość, nazywaną liczbą magiczną (ang. *magic number*), wtedy system traktuje plik jako program). Drugi parametr, `arg0`, jest zwyczajowo nazwą programu albo poleceń, pozbawioną poprzedzającej nazwy ścieżki. Ten i pozostała zmenna liczba argumentów (`arg1` do `argn`) są dostępne dla wywoływanego programu, odpowiadając argumentom wiersza poleceń w powloce. W istocie, sama powłoka też wywołuje polecenia przez użycie jednej z funkcji `exec` w połączaniu z `fork`. Ponieważ lista argumentów jest dowolnej długości, musi być zakończona przez pusty wskaźnik oznaczający koniec listy.

Jak zwykle, krótki przykład jest wart tysiąca słów, przyjrzyjmy się więc jak następujący program używa `execl` do uruchomienia programu wydruku katalogu `ls`:

```
/* runls -- używa "execl" do uruchomienia ls */

#include <unistd.h>

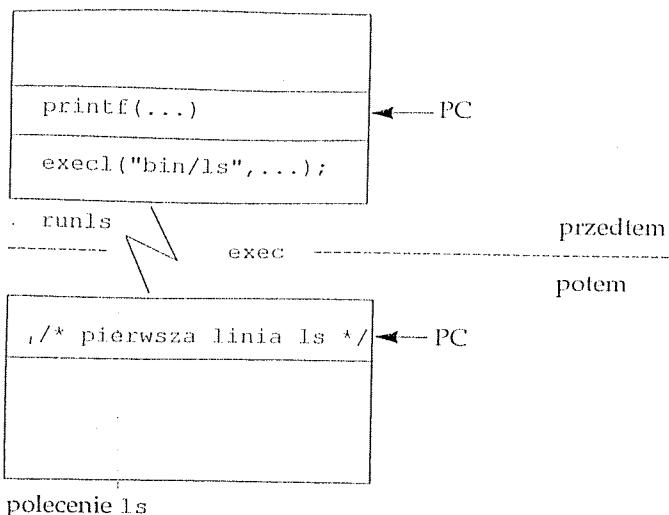
main()
{
    printf("executing ls\n");

    execl("/bin/ls", "ls", "-l", (char *)0);

    /* Jeżeli execl wróciła, wywołanie nie powiodło się, więc ... */
    perror("execl failed to run ls");
    exit (1);
}
```

Działanie tego przykładowego programu obrazuje rysunek 5.3. Część *przedtem* pokazuje proces bezpośrednio przed wywołaniem `execl`. Część *potem* pokazuje proces przekształcenia po wywołaniu `execl`, które teraz uruchomiło program `ls`. Licznik programu `PC` wskazuje na pierwszą linię `ls`, co oznacza, że `execl` powoduje rozpoczęcie wykonywania nowego programu od jego początku.

Zauważ, że w tym przykładzie po wywołaniu `execl` występuje bezwarunkowe wywołanie procedury bibliotecznej `perror`. Odzwierciedla to sposób pomyślnego wywołania `execl` (i wszystkie jego relacje), które wymazuje wywołujący program. Jeśli jednak wywołujący program pozostanie i `execl` powróci, wiadomo, że musiał wystąpić błąd. Jak z tego wynika, jeśli `execl` i jego pokrewne funkcje powracają zawsze zwracającą wartość `-1`.



Rysunek 5.3 Wywołanie funkcji exec

Funkcje execv, execvp i execvp

Inne formy funkcji exec dają programistę elastyczność budowy listy parametrów. Funkcja execv pobiera tylko dwa argumenty: pierwszy (path w powyższym opisie użycia) jest napisem zawierającym nazwę ścieżki programu, który ma być wykonywany. Drugi (argv) jest tablicą napisów, zadeklarowaną jako:

```
char * const argv[];
```

Pierwsza składowa tej tablicy wskazuje, ponownie jedynie zwyczajowo, na nazwę programu, który ma być wykonywany (z wyłączeniem nazwy ścieżki). Pozostałe składowe wskazują dodatkowe argumenty dla programu. Ponieważ lista ta jest nieokreślonej długości, zawsze musi być zakończona przez pusty wskaźnik.

Następny przykład używa execv do uruchomienia tego samego polecenia ls, co w poprzednim przykładzie:

```
/* runls2 -- używa execv do uruchomienia ls */

#include <unistd.h>

main ()
{
    char * const av[] = {"ls", "-l", (char *) 0};

    execv("/bin/ls", av);

    /* ponownie - uzyskanie tego oznacza błąd */
    perror("execv failed");
    exit(1);
}
```

Funkcje execlp i execvp są niemal identyczne jak execl i execv. Główna różnica polega na tym, że pierwszy argument dla execlp i execvp to po prostu nazwa pliku, a nie nazwa ścieżki. Przedrostek ścieżki dla tej nazwy pliku jest znajdowany przez przeszukiwanie katalogów wskazywanych przez zmienną środowiskową powłoki PATH. Oczywiście zmienna PATH może być łatwo ustawiona na poziomie powłoki za pomocą takich poleceń, jak:

```
$ PATH = /bin:/usr/bin:/usr/keith/mybin
$ export PATH
```

Oznacza to, że powłoka i execvp będą poszukiwać poleceń najpierw w /bin, później w /usr/bin i na końcu w /usr/keith/mybin.

Ćwiczenie 5.2 W jakich okolicznościach możesz użyć execv zamiast execl?

Ćwiczenie 5.3 Zalóżmy, że funkcje execvp i execlp nie istnieją. Napisz równoważny podprogram, używając execl i execv. Parametry dla tych procedur powinny składać się z listy katalogów i zestawu argumentów wiersza poleceń.

5.3.2 Dostęp do argumentów przekazanych za pomocą exec

Każdy program może uzyskać dostęp do argumentów w wywołaniu exec, przez parametry przekazywane do funkcji main programu. Mogą być one dostępne przy następującej definicji funkcji main programu:

```
main(int argc, char **argv)
{
    /* ciało programu */
}
```

Powinno to wyglądać znajomo, ponieważ ta sama technika jest używana dla dostępu do argumentów wiersza poleceń, który uruchomił program, co znów wskazuje, że sama powłoka używa exec w celu rozpoczęcia procesu. (W kilku poprzednich przykładach i ćwiczeniach zakładaliśmy znajomość parametrów wiersza poleceń. Jeśli ktoś miał jakiekolwiek problemy, ten podrozdział powinien wszyscy wyjaśnić).

W powyższej deklaracji funkcji main, argc jest całkowitym licznikiem liczb argumentów, natomiast argv wskazuje tablicę samych argumentów. Tak więc, jeśli program jest wykonywany za pomocą następującego wywołania execvp:

```
char * const argin[] = {"command", "with", "arguments", (char *) 0};
execvp("prog", argin);
wtedy wewnętrz prog obowiązują następujące warunki:
argc == 3
argv[0] == "command"
argv[1] == "with"
argv[2] == "arguments"
argv[3] == (char *) 0
```

Jako prosty przykład tej techniki rozważmy następny program, który drukuje swoje argumenty, oprócz pierwszego, na standardowym wyjściu:

```
/* myecho -- echo argumentów wiersza poleceń */

main (int argc, char **argv)
{
    while(--argc > 0)
        printf("%s ", *++argv);

    printf("\n");
}
```

Jeśli ten program jest wywoływany przez następujący fragment programu:

```
char * const argin[]={"myecho", "hello", "world", (char *)0};
execvp(argin[0], argin);
```

argc w myecho powinien być ustawiony na wartość 3 i wtedy zostanie wyświetlony następujący wynik:

```
hello world
```

który jest taki sam, jak przy poleceniu powłoki:

```
$ myecho hello world
```

Ćwiczenie 5.4 Napisz program waitcmd, który wykonuje dowolne polecenie, kiedy plik się zmienia. Powinien on pobierać nazwę obserwowanego pliku i polecenie do wykonania jako argumenty wiersza poleceń. Do monitorowania pliku mogą być używane funkcje stat i fstat. Program nie powinien niepotrzebnie marnować zasobów systemu; dlatego użyj standardowego podprogramu sleep (wprowadzonego w ćwiczeniu 2.16), w celu zrobienia przerwy w waitcmd i uzyskania rozsądniego odstępu po sprawdzeniu pliku. Jak powinien poradzić sobie ten program, jeśli plik początkowo nie istnieje?

5.4 Łączne użycie exec i fork

Połączone funkcje fork i exec oferują programistom potężne narzędzie. Za pomocą rozgałęzienia, a następnie użycia exec wewnętrznie niedawno utworzonego potomka, program może uruchomić inny program w ramach podprocesu i bez wymazywania samego siebie. Poniższy przykład pokazuje, jak to się dzieje. Wprowadzamy tu prostą procedurę błędu o nazwie fatal i funkcję systemową wait (raczej przedwcześnie). Funkcja wait (któram omówimy szczegółowo później) powoduje, że proces czeka, aż jego potomek zakończy pracę – jest to wspólne doświadczenie każdego rodzica.

```
/* runls3 -- uruchamia ls jako podproces */

#include <unistd.h>

main()
{
```

```
pid_t pid;

switch(pid = fork()) {
    case -1:
        fatal ("fork failed");
        break;
    case 0:
        /* proces potomny wywołuje exec */
        execl("/bin/ls", "ls", "-l", (char *)0);
        fatal("exec failed");
        break;
    default:
        /* proces rodzicielski używa wait w celu
           zawieszenia wykonania, dopóki
           proces potomny się nie zakończy */
        wait((int *)0);
        printf ("is completed\n");
        exit (0);
}
```

Procedura fatal po prostu używa perror do wyświetlenia komunikatu, a następnie wraca do systemu. (Instrukcja break po wywołaniu fatal zabezpiecza kod przed dalszymi zmianami podprogramu). Procedura fatal jest zaimplementowana następująco:

```
int fatal(char *s)
{
    perror(s);
    exit(1);
}
```

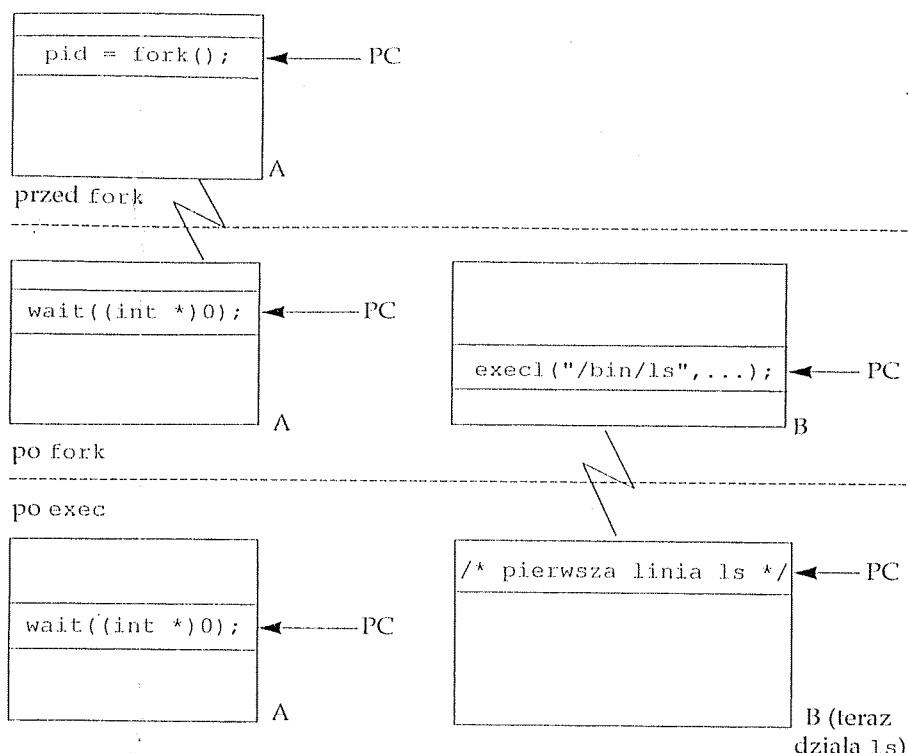
Ponownie użyjemy wykresu (rysunek 5.4) dla objaśnienia działania programu. Rysunek jest podzielony na trzy części: *przed fork*, *po fork* i *po exec*.

W początkowym stanie, *przed fork*, mamy pojedynczy proces A, a licznik programu PC wskazuje instrukcję fork, która jest następną instrukcją do wykonania.

Po wywołaniu fork mamy dwa procesy A i B. Proces rodzicielski A wykonuje funkcję systemową wait. Spowoduje to zawieszenie wykonania A, dopóki B się nie zakończy. Tymczasem B używa execl do załadowania polecenia ls.

Co stanie się później, pokazane jest w części *po exec* rysunku 5.4. Proces B został przekształcony i teraz wykonuje program ls. Licznik programu dla B został ustalony na pierwszą instrukcję ls. Ponieważ A czeka na zakończenie B; jego strzałka PC nie zmieniła pozycji.

Teraz powinieneś już ujrzeć zarys kilku mechanizmów używanych przez powłokę. Na przykład kiedy polecenie wykonywane jest w zwykły sposób, powłoka używa fork, exec i wait jak powyżej. Kiedy polecenie jest umieszczane w tle, zostaje pominięte wywołanie funkcji wait, a procesy powłoki i polecenia działają jednocześnie.



Rysunek 5.4. Połączone wywołanie funkcji fork i exec

Przykład docommand

Unix dostarcza procedurę biblioteczną o nazwie `system`, umożliwiającą wykonywanie polecenia powłoki z wnętrza programu. Używając `fork` i `exec`, zaimplementujemy szczegółową wersję tej procedury, o nazwie `docommand`. Skorzystamy z pośrednictwa standardowej powłoki (identyfikowanej przez nazwę ścieżki `/bin/sh`), zamiast prób bezpośredniego uruchamiania polecenia. Pozwoli to `docommand` wykorzystać zalety własności oferowanych przez powłokę, jak rozszerzanie nazwy pliku. Argument `-c` używany w wywołaniu powłoki powoduje pobranie polecenia z następnego argumentu napisowego, zamiast ze standardowego wejścia.

```
/* docommand -- uruchamia polecenie powłoki, wersja pierwsza */
```

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

```
int docommand(char *command)
{
    pid_t pid;
    if((pid = fork()) < 0)
```

```
        return(-1);

    if(pid == 0) /* potomek */
    {
        execl("/bin/sh", "sh", "-c", command, (char *)0);
        perror("execl");
        exit(1);
    }

    /* kod dla rodzica */
    /* oczekивание на завершение потомка */
    wait((int *)0);
    return(0);
}
```

Musimy tu powiedzieć, że jest to tylko pierwsze przybliżenie właściwej procedury bibliotecznej system. Na przykład, jeśli użytkownik końcowy programu naciśnie klawisz przerwania podczas wykonywania polecenia powłoki, polecenie i wywołujący program będą zatrzymane. Istnieje sposób ominięcia tego, ale odłożymy jego omówienie do następnego rozdziału.

5.5 Dziedziczenie danych i deskryptorów plików

5.5.1 Funkcja fork, pliki i dane

Tworzący za pomocą `fork` proces potomny jest niemal doskonałą kopią swojego rodzica. W szczególności wszystkie zmienne w procesie potomnym zachowają wartość, którą otrzymały w procesie rodzicielskim (jedynym wyjątkiem jest wartość zwracana przez samą funkcję `fork`). Ponieważ dane dostępne dla potomka są kopią danych dostępnych dla rodzica i zajmują inne bezwzględne miejsce w pamięci, trzeba uświadomić sobie, że późniejsze zmiany w jednym procesie nie będą wpływać na wartości zmiennych w drugim procesie.

Podobnie wszystkie pliki otwarte w procesie rodzicielskim są też otwarte w procesie potomnym; potomek utrzymuje swoją własną kopię deskryptorów plików związanych z każdym plikiem. Jednak po wywołaniu `fork` otwarte pliki pozostaną intymnie połączone w obu procesach. Dzieje się tak, ponieważ wskaźnik odczytu-zapisu dla każdego pliku jest współdzielony między potomkiem a rodzicem. Staje się to możliwe, ponieważ wskaźnik odczytu-zapisu jest utrzymywany przez system; nie zostaje włączony jawnie do samego procesu. W konsekwencji, kiedy proces potomny przesunię wskaźnik w pliku, proces rodzicielski również znajdzie się na nowej pozycji. Następujący krótki program pokazuje tę własność. Używamy w nim procedury `fatal`, wprowadzonej wcześniej w tym rozdziale, i nowej procedury o nazwie `printpos`. Dodatkowo zakładamy istnienie pliku o nazwie `data`, który ma co najmniej 20 znaków długości.

```
/* proc_file -- pokazuje obsługę plików przez forks */
/* zakłada, że "data" ma co najmniej 20 znaków długości */

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

```

main()
{
    int fd;
    pid_t pid;           /* process-id */
    char buf[10];        /* bufor dla danych pliku */

    if((fd = open ("data", O_RDONLY)) == -1)
        fatal("open failed");

    read(fd, buf, 10);      /* rozszerzony wskaźnik pliku */

    printpos("Before fork", fd);

    /* teraz tworzy dwa procesy */
    switch(pid = fork ()) {
    case -1:             /* błąd */
        fatal("fork failed");
        break;
    case 0:               /* potomek */
        printpos("Child before read", fd);
        read(fd, buf, 10);
        printpos("Child after read", fd);
        break;
    default:              /* rodzic */
        wait((int *)0);
        printpos("Parent after wait", fd);
    }
}

```

Procedura printpos po prostu wyświetla bieżącą pozycję w pliku razem z krótkim komunikatem. Może być ona zaimplementowana następująco:

```

/* drukuje pozycje w pliku */
int printpos(const char *string, int filedes)
{
    off_t pos;

    if(pos = lseek(filedes, 0, SEEK_CUR) == -1)
        fatal ("lseek failed");
    printf("%s:%ld\n", string, pos);
}

```

Kiedy uruchomiliśmy ten przykład, otrzymaliśmy następujące wyniki, dowodzące, że wskaźnik odczytu-zapisu jest współdzielony przez oba procesy:

```

Before fork: 10
Child before read: 10
Child after read:20
Parent after wait:20

```

Ćwiczenie 5.5 Napisz program, który pokazuje, że zmienne programu w procesie rodzicielskim i procesie potomnym mają te same wartości początkowe, ale są nawzajem niezależne.

Ćwiczenie 5.6 Sprawdź, co dzieje się wewnętrz procegu rodzicielskiego, jeśli proces potomny zamknie deskryptor pliku odziedziczony za pośrednictwem fork.

Czy dla procesu rodzicielskiego ten plik pozostanie otwarty, czy też zostanie zamknięty?

5.5.2 Funkcja exec i otwarte pliki

Otwarte deskryptory plików są zwykle przekazywane przez wywołanie funkcji exec. To znaczy, że plik otwarty w oryginalnym programie pozostaje otwarty, gdy exec rozpoczyna całkowicie nowy program. Wskaźnik odczytu-zapisu dla takiego pliku pozostaje niezmieniony przez wywołanie funkcji exec. (Oczywiście nie ma sensu mówić, że wartości zmiennych są chronione przez wywołanie exec, ponieważ oryginalny i nowo załadowany program są całkowicie różne).

Jednak można wykorzystać uniwersalną (dobrą na każdą pogodę) procedurę fcntl, aby ustawić związanego z plikiem znacznik close-on-exec. Jeśli zostanie on włączony (domyślnie jest wyłączony), wtedy plik będzie zamknięty przy wywołaniu jakiegokolwiek członka rodziny funkcji exec. Następujący fragment kodu pokazuje, jak włączyć znacznik close-on-exec:

```

#include <fcntl.h>

.

.

int fd;

fd = open("file", O_RDONLY);

.

.

/* ustaw znacznik close-on-exec */
fcntl(fd, F_SETFD, 1);

```

Znacznik close-on-exec może być wyłączony w ten sposób:

```
fcntl(fd, F_SETFD, 0);
```

Jego wartość może być uzyskana następująco:

```
res = fcntl(fd, F_GETFD, 0);
```

Zmienna całkowita res powinna być równa 1, jeśli znacznik close-on-exec jest włączony dla deskryptora fd, i równa 0 w przeciwnym przypadku.

5.6 Kończenie procesów za pomocą funkcji systemowej exit

Użycie

```

#include <stdlib.h>
void exit(int status);

```

Funkcja systemowa `exit` jest naszą dobrą znajomą, ale teraz możemy umieścić ją we właściwym kontekście. Używamy jej do końcaienia procesu, chociaż proces zatrzyma się także, gdy program osiągnie koniec funkcji `main` lub gdy funkcja `main` wykona instrukcję `return`.

Pojedynczy, całkowity argument `exit` nazywamy **stanem zakończenia** (ang. *exit status*) procesu. Jego młodsze 8 bajtów jest dostępne dla procesu rodzicielskiego pod warunkiem, że wykonywał funkcję systemową `wait` (więcej szczegółów podamy w następnym podrozdziale). Wartość zwracana w ten sposób przez `exit` zwykle jest używana do wskazania sukcesu albo niepowodzenia wykonywanego przez proces zadania. Zwyczajowo proces zwraca zero w przypadku normalnego zakończenia, a niezerową wartość w przypadku błędu.

Poza zatrzymaniem wywołującego procesu funkcja `exit` wykonuje pewną liczbę innych działań: co najważniejsze, zamknie wszystkie otwarte deskryptory plików. Jeśli proces rodzicielski wykonywał wywołanie `wait`, jak w ostatnim przykładzie, będzie ponownie uruchomiony. Funkcja `exit` może też wywoływać dowolną zdefiniowaną przez programistę procedurę obsługi zakończenia i wykonywać gruntowne działania czyszczące. Na przykład w Standardowej Bibliotece I/O mogą być one związane z buforowaniem. Programista może także ustawić co najmniej 32 procedury obsługi zakończenia, posługując się funkcją `atexit`.

Użycie

```
#include <stdlib.h>
int atexit(void (*func) (void));
```

Procedura `atexit` rejestruje funkcję wskazywaną przez `func`, wywoływaną bez żadnych parametrów. Wszystkie funkcje obsługi zakończenia zarejestrowane za pomocą `atexit`, przy zakończeniu będą wywoływanie w odwrotnej kolejności, niż były rejestrowane.

Dla porządku powinniśmy także wymienić funkcję systemową `_exit`, która różni się od `exit` początkowym podkreśleniem w nazwie. Jest ona używana w dokładnie ten sam sposób, co `exit`. Jednak omija opisane wcześniej gruntowne działania czyszczące. Dlatego większość programistów powinna unikać funkcji `_exit`.

Ćwiczenie 5.7 Stan zakończenia programu może być uzyskany za pomocą zmiennej `$?` powłoki, na przykład:

```
$ ls nonesuch
nonesuch: No such file or directory
$ echo $?
2
```

Napisz program o nazwie `fake`, który używa wartości całkowitej swojego pierwszego argumentu jako swojego stanu zakończenia. Używając naszkicowanej powyżej metody, wypróbuj `fake` używając rozmaitych argumentów, włącznie z ujemnymi i dużymi wartościami. Czy `fake` jest przydatnym programem?

5.7 Synchronizacja procesów

5.7.1 Funkcja systemowa `wait`

Użycie

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Jak widzieliśmy, funkcja `wait` chwilowo zawiesza wykonanie procesu, podczas gdy proces potomny działa. Gdy potomek zakończy się, czekający proces rodzicielski zostaje wznowiony. Jeśli działa więcej niż jeden proces potomny i gdy tylko którykolwiek z nich się zakończy, `wait` powraca.

Funkcja `wait` często jest wywołana przez proces rodzicielski zaraz po wywoaniu funkcji `fork`, na przykład:

```
int status;
pid_t cpid;

cpid = fork(); /* tworzy nowy proces */
if(cpid == 0) {
    /* potomek */
    /* wykonaj coś ... */
} else {
    /* rodzic, czeka na proces potomny */
    cpid = wait(&status);
    printf ("The child %d is dead\n", cpid);
}
```

To połączenie `fork` i `wait` jest najbardziej przydatne, gdy proces potomny ma uruchomić za pomocą wywołania `exec` całkiem różnych programów.

Wartość zwracana przez funkcję `wait` to zwykle process-id zakończonego potomka. Jeśli `wait` zwraca (`pid_t`) -1 może to oznaczać, że żaden proces potomny nie istnieje i w tym przypadku `errno` będzie zawierać kod błędu `ECHILD`. Aby stwierdzić, że każdy z procesów potomnych się zakończył, proces rodzicielski powinien w pętli czekać na zakończenie każdego potomka. Kiedy proces rodzicielski stwierdzi, że wszystkie procesy potomne zakończyły się, może podjąć dalsze działanie.

Funkcja `wait` pobiera jeden argument, `status`, który jest wskaźnikiem do liczby całkowitej. Jeśli wskaźnik jest równy `NULL`, wtedy argument jest po prostu ignorowany. Jednak jeśli do `wait` został przekazany ważny wskaźnik, po powrocie z wywołania `status` będzie zawierał użyteczną informację o stanie. Zwykle tą informacją będzie stan zakończenia procesu potomnego, przekazywany przez `exit`.

Następujący program `status` demonstruje, jak w tych okolicznościach używać funkcji `wait`:

```
/* status - jak uzyskać status wyjścia potomka */

#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
    pid_t pid;
    int status, exit_status;

    if((pid = fork()) < 0)
        fatal("fork failed");

    if(pid == 0)          /* potomek */
    {
        /* teraz wywołaj procedurę biblioteczną sleep,
         * aby zawiesić wykonanie na 4 sekundy */
        sleep(4);
        exit(5);           /* wyjście z wartością niezerową */
    }

    /* osiągniecie tego oznacza, że jest to rodzic */
    /* więc czeka na proces potomny */

    if((pid = wait(&status)) == -1)
    {
        perror("wait failed");
        exit(2);
    }

    /* test zakończenia potomka */
    if(WIFEXITED(status))
    {
        exit_status = WEXITSTATUS(status);
        printf("Exit status from %d was %d\n", pid, exit_status);
    }
}

exit(0);
}
```

Wartość zwracana do procesu rodzicielskiego przez `exit` jest zapamiętywana w 8 bardziej znaczących bitach zmiennej całkowitej `status`. W tym celu 8 mniej znaczących bitów musi być równe zeru. Makroinstrukcja `WIFEXITED` (zdefiniowana w `<sys/wait.h>`) testuje, czy jest tak rzeczywiście. Makroinstrukcja `WEXITSTA-`

TUS zwraca wartość zapamiętaną w bardziej znaczących bitach zmiennej `status`. Jeśli `WIFEXITED` zwraca 0, oznacza to, że proces potomny został zatrzymany przez inny proces, za pomocą metody komunikacji o nazwie `sygnał` (ang. `signal`). Sygnały będą omawiane w rozdziale 6.

Ćwiczenie 5.8 Dostosuj procedurę `docommand` tak, aby zwracała stan `exit` wykonywanego polecenia. Co mogło się zdarzyć, jeśli wywołanie `wait` zwróciło wartość -1?

5.7.2 Czekanie na konkretny proces potomny: `waitpid`

Funkcja systemowa `wait` pozwala procesowi rodzicielskiemu czekać na dowolnego potomka. Jednak jeśli rodzic chce być bardziej konkretny, może użyć funkcji systemowej `waitpid` do czekania na określony proces potomny.

Użycie

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Pierwszy argument, `pid`, określa process-id procesu potomnego, na który proces rodzicielski chce czekać. Jeśli jest on ustawiony na -1, a argument `options` na 0, wtedy `waitpid` zachowuje się dokładnie tak samo jak `wait`, ponieważ -1 wskazuje zainteresowanie każdym procesem potomnym. Jeśli `pid` jest większy od 0, wtedy proces rodzicielski będzie czekał na proces potomny z process- id równym `pid`. Drugi argument, `status`, po powrocie z `waitpid` będzie zawierał stan procesu potomnego.

Końcowy argument, `options`, może przybierać różne wartości zdefiniowane w `<sys/wait.h>`. Najbardziej przydatną z tych opcji jest `WNOHANG`. Pozwala ona na umieszczenie w pętli wywołania `waitpid` w celu monitorowania polożenia, ale nie blokuje działania procesu rodzicielskiego, jeśli proces potomny jeszcze działa. Gdy `WNOHANG` jest ustawione, a proces potomny jeszcze się nie zakończył, `waitpid` zwróci 0.

Funkcjonalność `waitpid` z opcją `WNOHANG` może być zademonstrowana przez modyfikację poprzedniego przykładu. Tym razem proces rodzicielski sprawdza, czy proces potomny się zakończył. Jeśli nie, wyświetla komunikat mówiący o tym, że jeszcze czeka, następnie zawiesza się na 1 sekundę i ponownie wywołuje `waitpid`, aby sprawdzić stan procesu potomnego. Zwróci uwagę, że proces potomny uzyskuje swoje process-id za pomocą wywołania `getpid`. Wyjaśnimy to w podrozdziale 5.10.1.

```
/* status2 --
 * jak uzyskać status wyjścia potomka za pomocą waitpid */
```

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
    pid_t pid;
    int status, exit_status;

    if((pid = fork()) < 0)
        fatal("fork failed");

    if(pid == 0)           /* potomek */
    {
        /* teraz wywołaj procedurę bibliotecną sleep
         * aby zawiesić wykonanie na 4 sekundy */
        printf("Child %d sleeping...\n", getpid());
        sleep(4);
        exit(5);           /* wyjście z wartością niezerową */
    }
    /* osiągnięcie tego oznacza, że jest to rodzic
     * zobacz więc, czy potomek wyszedł, oczekaj
     * 1 sekundę i powtórz test */
    while(waitpid(pid, &status, WNOHANG) == 0)
    {
        printf("Still waiting... \n");
        sleep(1);
    }

    /* test zakończenia potomka */
    if(WIFEXITED(status))
    {
        exit_status = WEXITSTATUS(status);
        printf("Exit status from %d was %d\n", pid, exit_status);
    }

    exit (0);
}
```

Po uruchomieniu program wyświetlił następujące komunikaty:

```
Still waiting...
Child 12857 sleeping...
Still waiting...
Still waiting...
Still waiting...
Exit status from 12857 was 5
```

5.8 Przedwczesne zakończenia i procesy zombie

Dotychczas przyjmowaliśmy, że funkcje `exit` i `wait` są używane w uporządkowany sposób, z oczekiwaniem na każdy podproces. Jednak mogą istnieć dwie inne, warte omówienia, sytuacje:

1. Proces potomny kończy się w czasie, gdy jego proces rodzicielski nie wykonuje funkcji `wait`.
2. Proces rodzicielski kończy się, gdy jeden lub więcej procesów potomnych ciągle działa.

W przypadku 1 kończący się proces jest umieszczany w stanie zawieszenia i staje się **zombie**. Jest to taki proces, który zajmuje pozycję w tabeli utrzymywanej w jądrze dla kontroli procesów, ale nie używa żadnych innych zasobów jądra. Zostanie on zakończony, jeśli jego proces rodzicielski zażąda potomka za pomocą wykonania funkcji `wait`. Proces rodzicielski będzie mógł wtedy odczytać stan zakończenia i pozycja stanie się dostępna dla ponownego użycia. W przypadku 2, proces rodzicielski może zakończyć się normalnie. Procesy potomne procesu rodzicielskiego (łącznie z zombie) są adoptowane przez proces `init` (process-id = 1).

5.9 Procesor poleceń: smallsh

Będziemy teraz budować prosty procesor poleceń o nazwie `smallsh`. Ten przykład ma dwie zalety. Po pierwsze, rozwija pojęcia, wprowadzone w tym rozdziale. Po drugie, demonstruje, że rzeczywiście nie ma nic szczególnego w standardowych poleceniach i narzędziach Uniksa. Pokazuje zwłaszcza, że powłoka jest tylko zwykłym programem, wywoływanym, gdy się zarejestrujesz w systemie.

Wymagania, które stawiamy `smallsh`, są proste. Program powinien gromadzić polecenia i wykonywać je w tle albo na pierwszym planie. Powinien też radzić sobie z liniami składającymi się z kilku polecień, rozdzielonych średnikami. Inne ułatwienia, takie jak rozszerzenie nazwy pliku lub przeadresowanie I/O, mogą być dodane później.

Elementarna logika wskazuje, że:

```
while(EOF nie wprowadzony)
{
    pobierz wiersz poleceń od użytkownika
    zgromadź argumenty polecień i wykonaj
    czekaj na proces potomny
}
```

Zaczniemy od funkcji `pobierz wiersz poleceń` o nazwie `userin`, która powinna drukować znak zachęty, a następnie czekać na linię wejściową z klawiatury. Każdy otrzymany znak wejściowy powinna umieścić w buforze programu. Zaimplementowaliśmy `userin` następująco:

```
/* plik włączany do przykładu */
#include "smallsh.h"

/* bufore i wskaźniki robocze programu */
static char inpbuf[MAXBUF], tokbuf[2*MAXBUF],
            *ptr = inpbuf, *tok = tokbuf;

/* drukuj znak zachęty i czytaj linię */
int userin(char *p)
```

```

{
    int c, count;

    /* inicjacja dla dalszych procedur */
    ptr = inpbuf;
    tok = tokbuf;

    /* wyświetl znak zachęty */
    printf ("%s", p);

    count = 0;

    while(1)
    {
        if((c = getchar()) == EOF)
            return(EOF);

        if(count < MAXBUF)
            inpbuf[count++] = c;

        if(c == '\n' && count < MAXBUF)
        {
            inpbuf[count] = '\0';
            return count;
        }

        /* restart, jeśli zbyt długa linia */
        if(c == '\n')
        {
            printf("smallsh: input line too long\n");
            count = 0;
            printf ("%s ",p);
        }
    }
}

```

Część ze szczegółów inicjacyjnych możemy na razie zignorować. Istotne jest, że userin najpierw drukuje znak zachęty (który jest przekazywany jako parametr), a następnie czyta znak po znaku od użytkownika, wracając, gdy napotka znak nowego wiersza albo koniec pliku (ten drugi przypadek jest wskazywany przez symbol EOF).

Używamy procedury `getchar` ze Standardowej Biblioteki I/O. Czyta ona pojedynczy znak ze standardowego wejścia programu, które zwykle odpowiada klawiturze. Funkcja `userin` umieszcza każdy nowy znak (jeśli jest to możliwe) w tablicy znaków `inpbuf`. Kiedy to ukończy, funkcja `userin` zwraca licznik wczytanych znaków lub EOF oznaczający koniec pliku. Zwróć uwagę, że znaki nowego wiersza są dodawane do `inpbuf`, a nie wyrzucane.

Włączany do `userin` plik `smallsh.h` zawiera definicje kilku przydatnych stałych (takich jak `MAXBUF`). Oto jego aktualna zawartość:

```

/* smallsh.h -- definicje dla procesora poleceń smallsh */

#include <unistd.h>

```

```

#include <stdio.h>
#include <sys/wait.h>

#define EOL      1      /* koniec linii */
#define ARG      2      /* zwykłe argumenty */
#define AMPERSAND 3
#define SEMICOLON 4

#define MAXARG   512   /* maks. liczba argumentów polecenia */
#define MAXBUF   512   /* maks. długość linii wejściowej */

#define FOREGROUND 0
#define BACKGROUND 1

```

Inne stałe, nie wykorzystywane w `userin`, spotkamy w dalszych procedurach. Plik `smallsh.h` włącza też standardowy plik nagłówkowy `<stdio.h>`. Daje nam to definicję `getchar` i stałą EOF.

Następna procedura, której się przyjrzymy, to `gettok`. Wylawia ona indywidualne leksemy (ang. *tokens*) z wiersza polecień zbudowanego przez `userin`. (Leksem jest najmniejszą jednostką językową mającą jakieś znaczenie, taką jak nazwa polecenia lub argument). Procedura `gettok` jest wywoływaną następująco:

```
toktype = gettok(&tptr);
```

Zmienna całkowita `toktype` zawiera wartość wskazującą rodzaj leksemu. Zakres możliwych wartości jest pobierany ze `smallsh.h` i zawiera EOL (koniec wiersza), SEMICOLON (średnik) itd. Parametr `tptr` jest wskaźnikiem znakowym, który wskazuje sam bieżący leksem po wywołaniu `gettok`. Ponieważ `gettok` będzie alokować własną pamięć dla ciągu znaków leksemu, musimy przekazać adres `tptr`, a nie jego wartość.

Kod źródłowy `gettok` znajduje się poniżej. Zauważ, że ponieważ odnosi się on do wskaźników znakowych `tok` i `ptr`, musi być zawarty w tym samym pliku źródłowym, co `userin`. (Teraz powinieneś już znać powód inicjacji `tok` i `ptr` na początku `userin`).

```

/* pobiera leksem, umieszcza w tokbuf */
int gettok(char **outptr)
{
    int type;

    /* ustaw napis outptr na tok */
    *outptr = tok;

    /* wytnij odstęp (białe spacje) z bufora zawierającego leksem */
    while(*ptr == ' ' || *ptr == '\t')
        ptr++;

    /* ustaw wskaźnik leksemu na pierwszy leksem w buforze */
    *tok++ = *ptr;

    /* ustaw typ zmiennej w zależności
     * od leksemu w buforze */
    switch(*ptr++) {

```

```

case '\n':
    type = EOL;
    break;

case '&':
    type = AMPERSAND;
    break;
case ';':
    type = SEMICOLON;
    break;
default:
    type = ARG;
    /* czytaj ważne zwykłe znaki */
    while(inarg(*ptr))
        *tok++ = *ptr++;
}

*tok++ = '\0';
return type;
)

```

Funkcja inarg umożliwia podjęcie decyzji, czy znak może być częścią zwykłego argumentu. Na razie musimy tylko sprawdzić, czy znak ma specjalne znaczenie dla smallsh, czy też nie:

```

static char special [] = {' ', '\t', '&', ';', '\n', '\0'};

int inarg(char c)
{
    char *wrk;

    for(wrk = special; *wrk; wrk++)
    {
        if(c == *wrk)
            return(0);
    }
    return(1);
)

```

Teraz możemy już wprowadzić funkcję, która wykonuje właściwą pracę. Funkcja procline używająca gettok będzie analizować składnię wiersza polecen, budując listę argumentów procesu. Kiedy natopka znak nowego wiersza albo średnik, wywoła procedurę o nazwie runcommand w celu wykonania polecenia. Zakłada ona, że linia wejściowa została już wczytana przez userin.

```

#include "smallsh.h"

int procline(void)          /* przetwarza linię wejściową */
{
    char *arg[MAXARG + 1]; /* tablica wskaźników dla runcommand */
    int toktype;           /* typ leksemu w poleceniu */
    int narg;               /* liczba dotychczasowych argumentów */
    int type;                /* FOREGROUND lub BACKGROUND */

    narg=0;

```

```

for(;;) /* pętla bez końca */
{
    /* podejmuje działanie w zależności od typu leksemu */
    switch(toktype = gettok(&arg[narg])){
    case ARG:
        if(narg < MAXARG)
            narg++;
        break;
    case EOL:
    case SEMICOLON:
    case AMPERSAND:
        if(toktype == AMPERSAND)
            type = BACKGROUND;
        else
            type = FOREGROUND;

        if(narg != 0)
        {
            arg[narg] = NULL;
            runcommand(arg, type);
        }

        if(toktype == EOL)
            return;

        narg = 0;
        break;
    }
}

```

Następnie musimy określić procedurę runcommand, która rzeczywiście zaczyna przetwarzanie polecenia. Zasadniczo jest ona modyfikacją używanej wcześniej procedury docommand. Zawiera dodatkowo parametr całkowity o nazwie where. Jeśli where jest ustawiony na wartość BACKGROUND, zdefiniowaną w smallsh.h, pomyślana jest wywołanie waitpid, a runcommand po prostu drukuje process-id i wraca.

```

#include "smallsh.h"

/* wykonuje polecenie z opcjonalnym czekaniem */
int runcommand(char **cline, int where)
{
    pid_t pid;
    int status;

    switch(pid = fork())
    {
    case -1:
        perror("smallsh");
        return(-1);
    case 0:
        execvp(*cline, cline);
        perror(*cline);
        exit(1);
    }

```

```
/* kod dla procesu rodzicielskiego */
/* jeśli proces w tle, drukuje pid i wychodzi */
if(where == BACKGROUND)
{
    printf("{Process id %d}\n", pid);
    return(0);
}

/* czeka, aż pid proces się zakończy */
if(waitpid(pid, &status, 0) == -1)
    return(-1);
else
    return(status);
}
```

Zauważ, że proste wywołanie `wait` z `doCommand` zostało zastąpione wywołaniem `waitpid`. Zapewnia to, że `runCommand` wyjdzie tylko wtedy, gdy ostatnio rozpoczęty proces potomny się zakończy i unika problemów z poleceniem w tle, które kończy się w międzyczasie. (Jeśli wydaje się to niejasne, pamiętaj, że `wait` zwraca process-id pierwszego zakończonego procesu potomnego, a nie ostatniego rozpoczętego).

Procedura `runCommand` wykorzystuje także funkcję systemową `execvp`. Zapewnia to, że program identyfikowany przez polecenie jest tropiony w ciągu katalogów bieżącej zmiennej środowiskowej `PATH`, chociaż w przeciwieństwie do prawdziwej powłoki, `smallsh` nie manipuluje zmiennej `PATH`.

Ostatni krok stanowi napisanie funkcji `main`, która wszystko powiąże. Jest to banalne ćwiczenie:

```
/* smallsh -- prosty procesor poleceń */

#include "smallsh.h"

char *prompt = "Command> "; /* znak zachęty */

main ()
{
    while(userin(prompt) != EOF)
        procline();
}
```

Ta procedura uzupełnia pierwszą wersję `smallsh`. Ponownie musimy podkreślić, że jest to tylko szkielet dowolnego skończonego rozwiązania. Podobnie jak w przypadku `doCommand`, własności `smallsh` są niedostateczne, gdy użytkownik wprowadzi bieżący znak przerwania, ponieważ spowoduje to zakończenie `smallsh`. Jak uodpornić naszą powłokę na takie sytuacje, zobaczymy w następnym rozdziale.

Ćwiczenie 5.9 Dodaj do `smallsh` mechanizm sekwencji unikowej (ang. *escape*) dla znaków specjalnych, jak ampersand (`&`) i średnik, tak, aby mogły one występuwać wewnętrz argumentów programu. Opracuj także poprawną interpretację komentarzy, wskazywanych przez początkowy znak hash (`#`). Co powinno się

działać ze znakiem zachęty, jeśli sekwencja unikowa będzie dotyczyła znaku nowego wiersza?

Ćwiczenie 5.10 Procedura `fcntl` może być używana do podwojenia deskryptora otwartego pliku. W tym kontekście jest ona wywoływana następująco:

```
newfdes = fcntl(filedes, F_DUPFD, reqvalue);
```

gdzie `filedes` to oryginalny deskryptor otwartego pliku. Stala `F_DUPFD` jest pobrana z systemowego pliku włączanego `<fcntl.h>`, natomiast `reqvalue` powinno być małą wartością całkowitą. Po pomyślnym wywołaniu zmienna `newfdes` powinna zawierać deskryptor pliku, który odnosi się do tego samego pliku, co `filedes` i ma tę samą wartość liczbową, co `reqvalue` (zakładając, że `reqvalue` nie jest deskryptorem pliku). Następujący fragment programu pokazuje, jak przekierować standardowe wejście, to jest deskryptor pliku 0.

```
fd1 = open ("somefile", O_RDONLY);
close(0);
fd2 = fcntl(fd1, F_DUPFD, 0);
```

Po wywołaniu wartość `fd2` będzie równa 0.

Używając tego wywołania w połączeniu z funkcjami systemowymi `open` i `close`, dostosuj `smallsh` tak, aby obsługiwał przekierowanie standardowego wejścia i wyjścia przy wykorzystaniu tej samej notacji, co konwencjonalna powłoka Uniksa. Pamiętaj, że standardowe wejście i wyjście odpowiada kolejno deskryptorom pliku 0 i 1.

Zwróć uwagę, że ta metoda podwajania deskryptorów plików może być także używana za pomocą funkcji systemowej `dup2`. (Dostępna jest także zbliżona funkcja `dup`).

5.10 Atrybuty procesu

Każdy proces Uniksa jest powiązany z pewną liczbą atrybutów, które pomagają systemowi kontrolować działanie i planowanie procesów, utrzymywać bezpieczeństwo systemu plików i tak dalej. Z tych atrybutów znamy już `process-id`, liczbę jednoznacznie identyfikującą proces. Inne atrybuty mają zasięg poczynański od środowiska, będącego zbiorem napisów utrzymywanych poza zdefiniowanym przez programistę obszarem danych, do faktycznego `user-id`, który określa przywileje systemu plików dla procesu. W pozostałej części tego rozdziału będziemy omawiać ważniejsze atrybuty procesu.

5.10.1 Process-id

Jak widzieliśmy, system nadaje każdemu procesowi nieujemny numer nazywany `process-id` (identyfikator procesu). `Process-id` jest zawsze unikatowy, chociaż będzie użyty ponownie, kiedy proces się zakończy. System rezerwuje kilka numerów `process-id` dla specjalnych procesów. Proces 0 (choć nazwany programem szeregującym albo zarządcą procesów) w rzeczywistości jest procesem wymiany.

Proces 1 to proces inicjacyjny, będący w rzeczywistości wykonaniem programu /etc/init. Proces 1 jest bezpośrednio lub pośrednio przodkiem każdego innego procesu w systemie Unix.

Program może otrzymywać swój własny process-id za pomocą następującego wywołania funkcji systemowej:

```
pid = getpid();
```

Podobnie, funkcja getppid zwraca process-id procesu rodzicielskiego dla procesu wywołującego:

```
ppid = getppid();
```

Następująca przykładowa procedura gentemp używa getpid do generowania unikatowej, tymczasowej nazwy pliku. Ta nazwa ma formę:

```
/tmp/tmp<pid>.<no>
```

Numer przyrostka jest zwiększany przy każdym wywołaniu gentemp. Procedura wywołuje także funkcję access, aby sprawdzić, czy plik już nie istnieje:

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

static int num = 0;

static char namebuf[20];
static char prefix[] = "/tmp/tmp";

char *gentemp(void)
{
    int length;
    pid_t pid;

    pid = getpid(); /* uzyskaj process-id */

    /* standardowe procedury obsługi napisów */
    strcpy(namebuf, prefix);
    length = strlen(namebuf);

    /* dodaj pid do nazwy pliku */
    itoa(pid, &namebuf[length]);

    strcat(namebuf, ". ");
    length = strlen(namebuf);

    do {
        /* dodaj numer przyrostka */
        itoa(num++, &namebuf[length]);
    } while(access(namebuf, F_OK) != -1);

    return(namebuf);
}
```

Procedura itoa po prostu zamienia liczbę całkowitą na równoważny napis:

```
/* itoa -- zamienia int na napis */

int itoa(int i, char *string)
{
    int power, j;

    j = i;

    for(power = 1; j >= 10; j /= 10)
        power *= 10;
    for(; power > 0; power /= 10)
    {
        *string++ = '0' + i/power;
        i %= power;
    }

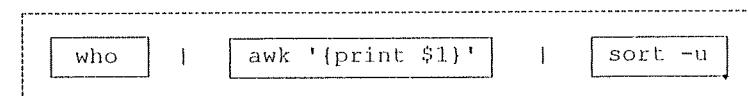
    *string = '\0';
}
```

Zwróć uwagę, że zamieniamy cyfrę na jej znakowy odpowiednik w pierwszej instrukcji wewnętrz drugiej pętli for. Powinieneś także zauważyc, że znacznie łatwiej większość powyższych działań może wykonać procedura sprintf. Opis sprintf znajdziesz w rozdziale 11.

Ćwiczenie 5.11 Zmodyfikuj procedurę gentemp tak, aby pobierała przedrostek tymczasowej nazwy pliku jako argument.

5.10.2 Grupy procesów i identyfikatory grup procesów

Unix pozwala na użyteczne umieszczanie procesów w grupach. Na przykład, gdy procesy są połączone za pomocą potoku z wiersza poleceń, najczęściej są umieszczane w grupie procesów. Rysunek 5.5 pokazuje taką typową grupę procesów, założoną przez wiersz poleceń powłoki.



Rysunek 5.5 Grupa procesów

Grupy procesów są przydatne, kiedy chcesz posługiwać się zestawem procesów jako całością, używając mechanizmu komunikacji międzyprocesowej nazywanego sygnałami, który omawiamy w rozdziale 6. Sygnał zwykle jest wysyłany do pojedynczego procesu, najczęściej powodując jego zakończenie, ale może być z łatwością wysyłany do całej grupy.

Każda grupa procesów jest wskazywana przez identyfikator grupy procesu (ang. *process group-id*) typu pid_t. Jeśli proces ma taki sam process-id, jak group-id pro-

cesu, uważany jest za wiodący proces grupy (i kiedy się kończy, podejmowanych jest kilka specjalnych działań). Początkowo proces dziedziczy swój identyfikator grupy procesu za pomocą `fork` albo `exec`.

Proces może uzyskać swój bieżący identyfikator grupy procesu za pomocą funkcji systemowej `getpgrp`:

Użycie

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp (void);
```

5.10.3 Zmiana grupy procesu

W powloce Uniksa, obsługującej kontrolę zadań, proces może chcieć umieścić się w nowej grupie procesów. Kontrola zadań pozwala powloce rozpoczęć wiele grup procesów (zadań) i kontrolować, które z grup procesów powinny pracować na pierwszym planie i dlatego mieć dostęp do terminala, a które powinny pracować w tle. Kontrola zadań jest rezyserowana za pomocą użycia sygnałów.

Proces może dołączyć do grupy lub utworzyć nową grupę procesów za pomocą wywołania funkcji systemowej `setpgid`:

Użycie

```
#include <sys/types.h>
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid);
```

Funkcja `setpgid` ustawia identyfikator grupy procesu dla procesu, który ma identyfikator `pid`, na wartość `pgid`. Jeśli `pid` zostanie ustawiony na 0, używany będzie identyfikator grupy procesu dla procesu *wywołującego*. Jeśli `pid` i `pgid` są takie same, wtedy proces staje się wiodącym procesem (przywódcą) grupy. W przypadku błędu zwracane jest -1. Jeśli `pgid` jest równy zeru, jako group-id procesu jest używany process-id wskazywany przez `pid`.

5.10.4 Sesje i identyfikatory sesji

Każda grupa procesów przynależy z kolei do sesji. Sesja jest w rzeczywistości podłączeniem procesów do terminala sterującego (ang. *controlling terminal*). Kiedy użytkownik rejestruje się, procesy i grupy procesów, które są tworzone jawnie lub niejawnie, będą należeć do sesji powiązanej z jego bieżącym terminaliem. Sesja jest najczęściej zbiorem pojedynczych grup procesów pierwzoplanowych używających terminala oraz jednej lub więcej grup procesów działających w tle. Sesja jest identyfikowana przez identyfikator sesji (ang. *session-id*) typu `pid_t`.

Proces może uzyskać swój bieżący session-id za pomocą następującego wywołania funkcji `getsid`:

Użycie

```
#include <sys/types.h>
#include <unistd.h>
pid_t getsid (pid_t pid);
```

Jeśli funkcji `getsid` przekazywana jest wartość 0, zwraca ona session-id procesu wywołującego, zaś w przeciwnym przypadku zwraca session-id procesu identyfikowanego przez `pid`.

Pojęcie sesji jest przydatne przy procesach działających w tle lub procesach demonach (ang. *daemons*). Demon to po prostu proces, który nie ma terminala sterującego. Przykładem jest `cron`, wykonujący polecenia w określonym momencie. Demon może znaleźć się w sesji *bez terminala sterującego* za pomocą wywołania funkcji `setsid` i przemieszczenia się do nowej sesji.

Użycie

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);
```

Jeśli wywołujący proces nie jest procesem wiodącym grupy, wtedy tworzona jest nowa grupa procesów i nowa sesja, a process-id procesu wywołującego staje się session-id. Nie będzie on także miał terminala sterującego. Demon znajdzie się wtedy w dziwnym stanie, będąc jedynym procesem w grupie procesów zawartej w nowej sesji, z wartością `pid` równą group-id procesu i session-id.

Funkcja `setsid` zawiedzie, jeśli proces wywołujący jest już procesem wiodącym grupy, i zwróci wtedy (`pid_t`) -1.

5.10.5 Środowisko

Środowisko (ang. *environment*) procesu to po prostu zbiór napisów zakończonych znakiem zerowym, reprezentowanych w programie jako zakończona pustym wskaźnikiem tablica wskaźników znakowych. Zwyczajowo każdy napis środowiska ma postać:

nazwa = coś

Programista może używać bezpośrednio środowiska w procesie przez dodawanie dodatkowego parametru `envp` do listy parametrów funkcji `main` wewnątrz programu. Następujący fragment programu pokazuje typ `envp`:

```
main(int argc, char **argv, char **envp)
{
    /* wykonaj coś */
}
```

Oto przykład trywialnego programu, drukującego tylko swoje środowisko:

```
/* showmyenv.c -- pokazuje środowisko */

main(int argc, char **argv, char **envp)
{
    while(*envp)
        printf("%s\n", *envp++);
}
```

Uruchomienie tego programu na jednym z komputerów dało następujący rezultat:

```
CDPATH=.:./
HOME=/usr/keith
LOGNAME=keith
MORE=-h -s
PATH=/bin:/etc:/usr/bin:/usr/cbin:/usr/lbin
SHELL=/bin/ksh
TERM=vt100
TZ=GMT0BST
```

Wydruk może wyglądać znajomo. Jest to środowisko procesu powłoki, który wywołał program `showmyenv`. Zawiera ono ważne zmienne używane przez powłokę, takie jak `HOME` i `PATH`.

Ten przykład pokazuje, że domyślne środowisko procesu jest takie samo jak procesu, który go utworzył za pomocą wywołania funkcji `exec` lub `fork`. Ponieważ środowisko jest przekazywane dalej w ten sposób, pozwala na prawie stałą rejestrację informacji, które inaczej musiałyby być ponownie określone dla każdego nowego procesu. Przydatności tego dowodzi zmienna środowiskowa `TERM`, pamiętająca bieżący typ terminala.

Aby określić nowe środowisko dla procesu, musisz użyć jednej z dwóch nowych funkcji z rodziny `exec`: `execle` i `execve`. Są one wywoływanie następująco:

```
execle(path, arg0, arg1, ..argn, (char *)0, envp);
i:
execve(path, argv, envp);
```

Powielają one działanie odpowiednio funkcji systemowych `execv` i `exec1`. Jedyna różnica polega na dodaniu parametru `envp`, którym jest zakończona pustym wskaźnikiem tablica wskaźników znakowych, określająca środowisko dla nowego programu. Następny przykład używa `execle` w celu przekazania nowego środowiska do programu `showmyenv`:

```
/* setmyenv.c - ustawia środowisko dla programu */
```

```
main ()
{
    char *argv[2], *envp[3];
```

```
argv[0] = "showmyenv";
argv[1] = (char *)0;

envp[0] = "foo=bar";
envp[1] = "bar=foo";
envp[2] = (char *)0;

execve("./showmyenv", argv, envp);

perror("execve failed");
}
```

Chociaż dopuszczalne jest użycie parametrów przekazywanych do funkcji `main`, preferowany sposób uzyskiwania przez proces dostępu do swojego środowiska stanowi zmienna globalna:

```
extern char **environ;
```

Do przejrzenia `environ` w poszukiwaniu nazwy zmiennej środowiskowej, występującej w formie nazwa = napis, może być użyta standardowa funkcja biblioteczna `getenv`.

Użycie

```
#include <stdlib.h>
char *getenv(const char *name);
```

Argumentem dla funkcji `getenv` jest tu nazwa zmiennej, którą pragniesz znaleźć. Jeśli poszukiwanie jest zakończone pomyślnie, `getenv` zwraca wskaźnik do części napisu stanowiącej wartość zmiennej, w przeciwnym przypadku zwraca wskaźnik `NULL`. Następujący kod pokazuje przykład jej użycia:

```
/* znajduje wartość zmiennej środowiskowej PATH */
```

```
#include <stdlib.h>
```

```
main()
```

```
{    printf ("PATH=%s\n", getenv("PATH")); }
```

Dostarczana jest także towarzysząca procedura `putenv`, która zmienia lub rozszerza środowisko. Jest ona wywoływaną za pomocą następującej linii:

```
putenv("NEWVARIABLE = value");
```

Procedura `putenv` zwraca zero, jeśli była wykonana pomyślnie. Zmienia ona środowisko wskazywane przez `environ`. Jednak nie zmienia wskaźnika `envp` w bieżącej funkcji `main`.

5.10.6 Bieżący katalog roboczy

Jak widzieliśmy w rozdziale 4, każdy proces jest związany z bieżącym katalogiem roboczym (ang. *current working directory*). Początkowe ustawienie dla bieżącego katalogu roboczego jest dziedziczone za pomocą funkcji fork lub exec, które rozpoczęły proces. Innymi słowy, proces zostaje początkowo umieszczony w tym samym katalogu, co jego proces rodzicielski.

Bardzo ważny jest fakt, że bieżący katalog roboczy stanowi atrybut procesu. Gdy proces potomny zmienia położenie za pomocą wywołania chdir (zdefiniowanego w rozdziale 4), bieżący katalog roboczy procesu rodzicielskiego pozostaje niezmieniony. Z tego powodu standardowe polecenie cd jest faktycznie polecienniem wbudowanym w samą powłokę i nie ma odpowiednika programowego.

5.10.7 Bieżący katalog główny

Każdy proces jest też powiązany z katalogiem głównym, używanym w bezwzględnych nazwach ścieżek poszukiwania. Podobnie jak w przypadku bieżącego katalogu roboczego, katalog główny procesu jest początkowo określony przez jego proces rodzicielski. Unix dostarcza funkcję systemową o nazwie chroot w celu zmiany początku hierarchii systemu plików dla procesu.

Argument path wskazuje nazwę ścieżki zawierającą katalog. Jeśli wywołanie funkcji chroot zakończy się pomyślnie, path stanie się punktem startowym dla tych poszukiwań plików, które zaczynają się od / (tylko dla procesu wywołującego, reszty systemu to nie dotyczy). Funkcja chroot zwraca -1 i katalog główny pozostaje niezmieniony, jeśli wywołanie zawiedzie. Proces wywołujący musi mieć właściwe przywileje zmiany katalogu głównego.

Użycie

```
#include <unistd.h>
int chroot(const char *path);
```

Ćwiczenie 5.12 Dodaj polecenie cd do procesora poleceń smallsh.

Ćwiczenie 5.13 Napisz własną wersję funkcji getenv.

5.10.8 Identyfikatory użytkownika i grupy

Każdy proces jest powiązany z rzeczywistym identyfikatorem użytkownika (ang. *real user-id*) i rzeczywistym identyfikatorem grupy (ang. *real group-id*). Jest to zawsze user-id i bieżący group-id dla użytkownika, który powołał proces.

Przy rozstrzyganiu, czy proces może uzyskać dostęp do pliku, używane są efektywny user-id i group-id. Najczęściej są one takie same jak rzeczywisty user-id

i group-id. Jednak proces albo któryś z jego przodków może mieć ustawiony swój bit set-user-id lub set-group-id. Na przykład, jeśli plik programu ma ustawiony bit set-user-id, gdy program jest wywoływany za pomocą wywołania funkcji exec, efektywnym user-id procesu staje się user-id właściciela pliku, a nie użytkownika, który rozpoczął proces.

Istnieje kilka funkcji systemowych, dostępnych w celu uzyskania powiązanych z procesem user-id i group-id. Następujący fragment programu pokazuje ich zastosowania:

```
#include <unistd.h>

main()
{
    uid_t uid, euid;
    gid_t gid, egid;

    /* pobiera rzeczywisty user-id */
    uid = getuid();

    /* pobiera efektywny user-id */
    euid = geteuid();

    /* pobiera rzeczywisty group-id */
    gid = getgid();

    /* pobiera efektywny group-id */
    egid = getegid();
```

Dostępne są też dwie funkcje dla ustawiania efektywnego user-id i group-id dla procesu:

```
#include <unistd.h>

uid_t newuid;
gid_t newgid;

/* ustawia efektywny user-id */
status = setuid(newuid);

/* ustawia efektywny group-id */
status = setgid(newgid);
```

Proces rozpoczęty przez nieuprzywilejowanego użytkownika (to znaczy kogoś, kto nie jest super-użytkownikiem) może tylko przywrócić swojemu user-id i group-id rzeczywiste wartości. Super-użytkownik jak zwykle ma wolną rękę. Wartością zwracaną przez obie procedury jest 0 przy pomyślnym zakończeniu i -1 w przypadku błędu.

Ćwiczenie 5.13 Napisz procedurę, która uzyskuje rzeczywiste user-id i group-id procesu wywołującego, a następnie zapisuje ich odpowiednik ASCII na końcu pliku dziennika.

5.10.9 Ograniczenie wielkości pliku: funkcja ulimit

Istnieje przypadające na proces ograniczenie wielkości pliku, który może być utworzony za pomocą funkcji systemowej `write`. To ograniczenie dotyczy także sytuacji, gdy rozszerzany jest istniejący plik, mniejszy niż ograniczenie.

Ograniczenie wielkości pliku może być zmieniane za pomocą funkcji systemowej `ulimit`.

Użycie

```
#include <ulimit.h>
long ulimit(int cmd, {long newlimit});
```

Aby uzyskać bieżącą wartość ograniczenia wielkości pliku, programista może wywołać funkcję `ulimit` z parametrem `cmd` ustawionym na `UL_GETFSIZE`. Zwrócona wartość jest wyrażona w 512-bajtowych blokach.

Aby zmienić ograniczenie wielkości pliku, programista powinien ustawić `cmd` na `UL_SETFSIZE` i umieścić nowe ograniczenie wielkości pliku, znowu w 512-bajtowych blokach, w `newlimit`, na przykład:

```
if(ulimit(UL_SETFSIZE, newlimit) < 0)
    perror("ulimit failed");
```

W rzeczywistości tylko super-użytkownik może powiększać w ten sposób ograniczenie wielkości pliku. Jednak procesy, które mają faktyczny user-id innych użytkowników, mogą zmniejszać to ograniczenie.

5.10.10 Priorytety procesów: nice

System decyduje o proporcji czasu CPU przydzielanej konkretnemu procesowi częściowo na podstawie wartości całkowitej `nice`. Wartość `nice` ma zakres od 0 do zależnego od systemu maksimum. Im wyższa liczba, tym niższy priorytet procesu. Spolecznie świadomy proces może zmniejszać swój priorytet i w ten sposób przydzielać więcej zasobów innym procesom, używając funkcji systemowej `nice`. Pobiera ona jeden argument, liczbę dodawaną do aktualnej wartości `nice`, na przykład:

```
nice(5);
```

Super-użytkownik (i tylko super-użytkownik) procesu może powiększyć jego priorytet przez użycie ujemnej wartości, jako parametru wywołania `nice`. Funkcja `nice` jest przydatna jeśli na przykład chcesz obliczyć wartość π z dokładnością do stu milionów miejsc po przecinku i nie wpływać na czas działania systemu dla innych użytkowników. Funkcja `nice` jest starym wywołaniem. Ostatnie opcjonalne rozszerzenia czasu rzeczywistego w POSIX-ie umożliwiają znacznie bardziej precyzyjną kontrolę. Jednak wykraczają one poza zakres tej książki, więc nie będziemy ich tu rozważać.

ROZDZIAŁ 6

Sygnały i przetwarzanie sygnałów

- 6.1 Wprowadzenie
- 6.2 Obsługa sygnałów
- 6.3 Blokowanie sygnałów
- 6.4 Wysyłanie sygnałów

6.1 Wprowadzenie

Często pożądana jest budowa systemu oprogramowania, składającego się z kilku współpracujących ze sobą procesów, a nie tylko z pojedynczego, monolitycznego programu. Istnieje wiele powodów takiego podejścia: na przykład pojedynczy program może być zbyt duży pod względem wielkości wymaganej fizycznej pamięci lub dostępnej przestrzeni adresowej, aby mógł pracować w komputerze; część wymaganych funkcji mogą już realizować istniejące programy. Taki problem najlepiej rozwiązać za pomocą procesu serwera, współpracującego z dowolną liczbą procesów klienta; można również wykorzystać wiele procesów i tak dalej.

Na szczęście Unix jest bogato wyposażony w mechanizmy komunikacji międzyprocesowej. W tym i następnych rozdziałach będziemy omawiać trzy najszerzej używane własności: **sygnały**, **potoki** i **FIFO**. Razem z bardziej złożonymi własnościami, które objaśnimy w rozdziałach 8 i 10, stwarzają one duże możliwości tworząc oprogramowania, który chce budować systemy wieloprocesowe.

Zacznijmy od przyjrzenia się sygnałom.

Przypuśćmy, że uruchomisz polecenie Uniksa, które wygląda na długotrwałe:

```
$ cc verybigprog.c
```

Następnie stwierdzasz, że coś przebiega nieprawidłowo i polecenie ostatecznie nie powiedzie się. Aby oszczędzić czas, możesz zatrzymać polecenie przez wciśnięcie bieżącego klawisza **przerwania** (ang. *interrupt*), którym często jest [Del] albo [Ctrl+C]. Polecenie zostanie zakończone i powrócisz do znaku zachęty powłoki.

Spójrzmy teraz, co się dzieje w rzeczywistości: część jądra odpowiedzialnego za wejście z klawiatury zobaczy znak przerwania. Wtedy jądro wyśle sygnał o nazwie `SIGINT` do wszystkich procesów, które rozpoznają terminal jako swój terminal ste-

rujący. Dotyczy to również wywołania `cc`. Kiedy `cc` odbierze sygnał, wykona domyślne działanie związane z `SIGINT` i zakończy się. Ciekawe, że proces powłoki związany z terminaliem też odbiera `SIGINT`. Ponieważ musi jednak pozostać, by interpretować późniejsze polecenia, praktycznie ignoriuje ten sygnał. Jak zobaczymy, programy też mogą wybrać przechwycenie `SIGINT`, co oznacza, że ilekroć użytkownik nacisnął klawisz przerwania, wykonują specjalną procedurę obsługi przerwania.

Sygnalów używa też jądro, aby uporać się z pewnymi rodzajami poważnych błędów. Na przykład założymy, że plik programu został w pewien sposób uszkodzony i zawiera nielegalne instrukcje maszynowe. Kiedy proces będzie wykonywał program, jądro dostrzeże próbę wykonania nielegalnych instrukcji i wyśle sygnał `SIGILL` (ang. *illegal*), aby go zakończyć. Nawiązany dialog może wyglądać tak:

```
$ badprog
Illegal instruction - core dumped
```

Znaczenie wyrażenia „zrzut rdzenia” (ang. *core dumped*) wyjaśnimy później.

W podobny sposób jak z jądra do procesu, sygnały mogą być również wysyłane od jednego procesu do drugiego. Najprościej pokazać to za pomocą polecenia `kill`. Na przykład przypuśćmy, że programista uruchomił długotrwałe polecenie w tle:

```
$ cc verybigprog.c &
[1] 1098
```

i później zdecydował się je zakończyć. Wtedy może użyć polecenia `kill` do wysłania sygnału `SIGTERM` do procesu. Podobnie jak `SIGINT`, `SIGTERM` kończy proces, jeśli nie ma innych jawnych ustaleń. Jako argument polecenia `kill` musi być podana wartość process-id:

```
$ kill 1098
Terminated
```

Sygnały dostarczają prostej metody przekazywania przerwań programowych do procesu uniksu. Jeśli potrzebujesz metafory, pomyśl o sygnale jako rodzaju programowego zaworu, który przerywa proces, niezależnie od tego, co on aktualnie robi. Z powodu swojej natury, sygnały są używane raczej do obsługi nietypowych sytuacji, a nie do prostego przesyłania danych pomiędzy procesami.

Krótko mówiąc, proces może zrobić z sygnałem następujące trzy rzeczy:

1. Wybrać sposób reakcji po otrzymaniu konkretnego sygnału (obsługa sygnałów).
2. Blokować sygnały (to znaczy pozostawić je na później) w określonych fragmentach krytycznego kodu.
3. Wysłać sygnał do innego procesu.

W pozostałej części rozdziału przyjrzymy się dokładniej każdej z tych sytuacji.

6.1.1 Nazwy sygnałów

Sygnały nie mogą bezpośrednio przenosić informacji, co ogranicza ich użyteczność jako ogólnego mechanizmu komunikacji międzyprocesowej. Jednak każdy typ sygnału ma nadaną mnemoniczną nazwę (przykładem może tu być `SIGINT`), wskazującą cel, w jakim ten sygnał jest zwykle używany. Nazwy sygnałów zdefiniowane są w standardowym pliku nagłówkowym `<signal.h>` za pomocą dyrektywy preprocessora `#define`. Jak możesz oczekwać, te nazwy oznaczają małe, dodatnie liczby całkowite. Na przykład sygnał `SIGINT` jest zwykle zdefiniowany jako:

```
#define SIGINT 2 /* przerwanie (klawisz wymazania) */
```

Większość dostarczanych przez Unix typów sygnałów jest przeznaczona do użycia przez jądro, chociaż istnieje również kilka do przesyłania z procesu do procesu. Poniżej zamieszczamy kompletną listę standardowych sygnałów, jakie opisuje *XSI*, i ich znaczenie. Lista sygnałów jest ulożona w kolejności alfabetycznej w celu ułatwienia poszukiwania. Przy pierwszym czytaniu możesz tę listę spokojnie opuścić.

- **SIGABRT** Sygnał przerwania procesu (ang. *process abort signal*); wysyłany przez bieżący proces za pomocą wywołania funkcji `abort`. `SIGABRT` powinien zakończyć się tym, co *XSI* opisuje jako **zakończenie anormalne** (ang. *abnormal termination*). Rzeczywisty skutek w implementacjach Uniksa stanowi **zrzut rdzenia** (ang. *core dump*), wskazywany przez komunikat `Quit - core dumped`, gdzie obraz procesu jest zrzucany do pliku dyskowego w celu późniejszej analizy. Więcej na ten temat powiemy później.
- **SIGALRM** Zegar alarmu (ang. *alarm clock*); wysyłany przez jądro do procesu, gdy upłynie ustalony czas. Wszystkie procesy mogą ustawić co najmniej trzy czasomierze (ang. *interval timers*). (Pierwszy z nich może mierzyć rzeczywisty upływ czasu). Czasomierz jest ustawiany przez sam proces za pomocą wywołania funkcji systemowej `alarm` (lub ustawienia pierwszego parametru rzadziej stosowanej funkcji systemowej `setitimer` na `ITIMER_REAL`). Funkcję `alarm` opisujemy w podrozdziale 6.4.2. Jeśli chcesz wiedzieć więcej o funkcji `setitimer`, sprawdź w swoim lokalnym podręczniku.
- **SIGBUS** Błąd szyny (ang. *bus error*); wysyłany, jeśli została dostrzeżona usterka sprzętowa. Błąd szyny jest zdefiniowany w implementacji i podobnie jak `SIGABRT` powoduje anormalne zakończenie.
- **SIGCHLD** Proces potomny zakończony lub zatrzymany (ang. *child process terminated or stopped*). Ilekroć proces potomny kończy się lub zatrzymuje, jądro zawiadamia jego proces rodzicielski, wysyłając mu `SIGCHLD` – sygnał śmierci potomka. Domyślnie proces rodzicielski ignoriuje sygnał, więc jeśli pragnie wiedzieć o każdym ukończonym procesie potomnym, musi jawnie przechwytywać sygnał `SIGCHLD`.
- **SIGCONT** Kontynuuj proces, jeśli był zatrzymany (ang. *continue executing if stopped*). Jest to sygnał sterowania pracą, który powinien wznowić proces, jeśli został on zatrzymany; w przeciwnym przypadku proces powinien ignorować sygnał. Stanowi on przeciwieństwo sygnału `SIGSTOP`.

- SIGFPE Wyjątek zmiennoprzecinkowy (ang. *floating-point exception*); wysyłany przez jądro, gdy wystąpi błąd obliczeń zmiennoprzecinkowych (taki jak nadmiar lub niedomiar). Powoduje on anormalne zakończenie.
- SIGHUP Sygnal zawieszenia (ang. *hangup signal*); wysyłany przez jądro do wszystkich procesów powiązanych z **terminaliem sterującym**, jeśli zostaje on odłączony. (Najczęściej terminaliem sterującym grupy procesów jest własny terminal użytkownika. Pojęcie to omawiamy szerzej w rozdziale 9). Zostaże on także wysłany do wszystkich członków sesji, gdy kończy się proces wiodący sesję, którym zwykle jest proces powłoki, pod warunkiem, że sesja jest powiązana z terminaliem sterującym. Dzięki temu, gdy użytkownik się wyrejestruje, procesy działające w tle są kończone, jeśli jawne ustalenia nie stanowią inaczej (więcej szczegółów znajdziesz w podrozdziale 5.10).
- SIGILL Nielegalna instrukcja (ang. *illegal instruction*); wysyłany przez system operacyjny, gdy proces próbuje wykonać nielegalną instrukcję. Staje się to możliwe, kiedy program uszkodzi swój własny kod, w istocie jest to jednak nieprawdopodobne. Innym powodem może być próba wykonania instrukcji zmiennoprzecinkowej bez właściwego wsparcia sprzętowego, co wydaje się bardziej prawdopodobne. SIGILL powoduje anormalne zakończenie.
- SIGINT Przerwanie (ang. *interrupt*); wysyłany przez jądro do wszystkich procesów powiązanych z sesją terminala, gdy użytkownik wcisnie klawisz przerwania. Jest to powszechnie stosowany sposób zatrzymania działającego programu.
- SIGKILL Usunięcie (ang. *kill*); specjalny sygnał, wysyłany od jednego procesu do innego, aby usunąć odbiorcę. Jest on także czasami wysyłany przez system (podczas zamknięcia systemu). SIGKILL to jeden z dwóch sygnałów, które nie mogą być zignorowane lub przechwycone (to znaczy obsłużone przez procedurę przerwania zdefiniowaną przez użytkownika).
- SIGPIPE Zapis do potoku lub gniazda, kiedy odbiorca zakończy się (ang. *write on a pipe or socket, when recipient has terminated*). Potoki i gniazda są innymi mechanizmami komunikacji międzyprocesowej, które omówimy w późniejszych rozdziałach. Również SIGPIPE będzie wtedy omówiony bliżej.
- SIGPOLL Zdarzenie odpytane (ang. *pollable event*); sygnał generowany przez jądro, kiedy otwarty deskryptor pliku jest gotowy do operacji wejścia lub wyjścia. Jednak łatwiejszy sposób implementacji odpytania stanowi użycie funkcji systemowej `select`, którą opisujemy szczegółowo w rozdziale 7.
- SIGPROF Upływał czas profilowania (ang. *profiling time expired*). Jak wspomniano powyżej przy sygnale SIGALRM, wszystkie procesy mogą ustawić co najmniej trzy czasomierze. Drugi z nich może być ustawiony na pomiar czasu wykonywania procesu w trybie jądra i trybie użytkownika. Sygnał SIGPROF jest generowany, gdy upłynie ustalony czas i dletoż może być używany przez interpretery do profilowania wykonania programu. (Czasomierz jest ustawiany za pomocą ustawienia pierwszego parametru funkcji `setitimer` na `ITIMER_PROF`).

- SIGQUIT Zakończenie (ang. *quit*). Bardzo podobny do SIGINT; wysyłany przez jądro, kiedy użytkownik wcisnie klawisz zakończenia, powiązany z jego terminaliem. Wartością domyślną klucza zakończenia jest ASCII FS lub [Ctrl+\]. W odróżnieniu od SIGINT, powoduje anormalne zakończenie i zrzut zawartości rdzenia.
- SIGSEGV Nieważna referencja do pamięci (ang. *invalid memory reference*). SEGV oznacza pogwałcenie segmentacji i jest generowany, kiedy proces próbuje uzyskać dostęp do nieważnego adresu pamięci. SIGSEGV powoduje anormalne zakończenie.
- SIGSTOP Zatrzymanie wykonania (ang. *stop executing*); sygnał kontroli zadań, który zatrzymuje proces. Podobny do SIGKILL w tym, że nie może być przechwycony lub zignorowany.
- SIGSYS Nieważne wywołanie funkcji systemowej (ang. *invalid system call*); wysyłany przez jądro, jeśli proces próbuje wykonać instrukcję maszynową, która nie jest wywołaniem funkcji systemowej. Jest to kolejny sygnał, który powoduje anormalne zakończenie.
- SIGTERM Programowy sygnał zakończenia (ang. *software termination signal*). Zwyczajowo jest używany do kończenia procesu (co można wywnioskować z jego nazwy). Programista może użyć tego sygnału, aby dać procesowi trochę czasu na działania porządkujące przed wysłaniem sygnału SIGKILL.
- SIGTRAP Pułapka śladu (ang. *trace trap*); specjalny sygnał używany przez programy uruchomieniowe (debugery), takie jak sdb i adb, w połączeniu z funkcją systemową `ptrace`. Z powodu jego specjalistycznej natury, nie będziemy omawiać go dalej. Domyślnie SIGTRAP powoduje anormalne zakończenie.
- SIGTSTP Sygnał zatrzymania terminala (ang. *terminal stop signal*); generowany przez użytkownika wciskającego klawisz zawieszenia (zwykle [Ctrl+Z]). Sygnał SIGTSTP jest podobny do SIGSTOP, jednak może być przechwytywany i ignorowany.
- SIGTTIN Proces działający w tle próbuje odczytu (ang. *background process attempting read*); wysyłany, ilekroć proces jest wykonywany w tle i próbuje odczytu ze swojego terminala sterującego. Domyślnym działaniem jest zatrzymanie procesu.
- SIGTTOU Proces działający w tle próbuje zapisu (ang. *background process attempting write*); zbliżony do SIGTTIN z tym wyjątkiem, że jest generowany, kiedy proces wykonywany w tle próbuje zapisu do terminala sterującego. I podobnie jak w przypadku SIGTTIN, domyślnym działaniem jest zatrzymanie procesu.
- SIGURG W gnieździe dostępne szerokopasmowe dane (ang. *high bandwidth data is available at a socket*); mówi procesowi, że za pomocą połączenia sieciowego zostały otrzymane pilne lub pozapasmowe dane.
- SIGUSR1 i SIGUSR2 Podobnie jak SIGTERM, te sygnały nigdy nie są wysyłane przez jądro i mogą być wykorzystane przez użytkownika w dowolnym celu.

- SIGVTALRM Upłynął czas wirtualnego czasomierza (ang. *virtual timer expired*). Jak wspominano przy SIGALRM i SIGPROF, wszystkie procesy mają co najmniej trzy czasomierze. Ostatni z nich może być ustawiony na pomiar czasu wykonywania procesu w trybie użytkownika. (Czasomierz jest ustawiany za pomocą ustawienia pierwszego parametru funkcji setitimer na TTIMER_VIRTUAL).
- SIGXCPU Przekroczyony limit czasu CPU (ang. *CPU time limit exceeded*); generowany, jeśli proces przekroczy swój maksymalny limit czasu CPU. Domyślnie powoduje anormalne zakończenie.
- SIGXFSZ Przekroczyony limit wielkości pliku (ang. *file size limit exceeded*); generowany, jeśli proces przekroczy swój maksymalny limit wielkości pliku. Domyślnie powoduje anormalne zakończenie.

Istnieje pewna liczba innych sygnałów, zależnych od implementacji i znajdujących się poza obszarem XSI. Większość z nich jest używana przez jądro do wskazywania warunków błędu, na przykład: SIGEMT (pułapka emulatora – ang. *emulator trap*) często wskazuje zdefiniowaną przez implementację usterkę sprzętową.

6.1.2 Normalne i anormalne zakończenie

Po otrzymaniu większości sygnałów następuje normalne zakończenie procesu. Efekt jest mniej więcej taki sam, jak gdyby proces wykonał zaimprowizowane wywołanie funkcji _exit. Zwracany w takich okolicznościach do procesu rodzicielskiego stan wyjścia mówi mu, co się zdarzyło. W <sys/wait.h> zdefiniowane są makroinstrukcje, pozwalające procesowi rodzicielskiemu określić powód zakończenia i w tym konkretnym przypadku wartość sygnalu, który jest za to odpowiedzialny. Następujący fragment kodu pokazuje proces rodzicielski, testujący powód zakończenia i drukujący stosowny komunikat.

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

if((pid=wait(&status)) == -1)
{
    perror("wait failed");
    exit(1);
}

/* testuj, czy proces potomny zakończył się normalnie */
if(WIFEXITED(status))
{
    exit_status = WEXITSTATUS(status);
    printf("Exit status from %d was %d\n", pid, exit_status);
}

/* testuj, czy proces potomny odebrał sygnał */
if(WIFSIGNALED(status))
{
    sig_no = WTERMSIG(status);
    printf("Signal number %d terminated child %d\n", sig_no, pid);
}
```

```
/* testuj, czy proces potomny został zatrzymany */
if(WIFSTOPPED(status))
{
    sig_no = WSTOPSIG(status);
    printf("Signal number %d stopped child %d\n", sig_no, pid);
}
```

Jak widzieliśmy powyżej, sygnały SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP, SIGSYS, SIGXCPU, SIGXFSZ i SIGFPE powodują anormalne zakończenie i zwykle skutkiem tego jest zrzut rdzenia. Oznacza to, że zrzut zawartości pamięci procesu jest zapisywany do pliku o nazwie core w bieżącym katalogu roboczym procesu (rdzeń (ang. *core*) to oczywiście raczej archaiczny sposób opisu pamięci głównej). Plik rdzenia będzie zawierał, w formie dwójkowej, wartości wszystkich zmiennych programu, rejestrów sprzętowych i informacje kontrolne jądra z chwilą, gdy nastąpiło zakończenie. Stan wyjścia procesu, który zakończył się anormalnie, będzie taki sam, jak przy normalnym zakończeniu za pomocą sygnalu, z wyjątkiem tego, że ustawiony jest w nim siódmy mniej znaczący bit. Większość systemów Unixa definiuje obecnie makroinstrukcję WCOREDUMP, która zwraca prawdę albo fałsz w zależności od tego, czy odpowiedni bit w zmiennej status jest ustawiony. Uważaj jednak, ponieważ makroinstrukcja WCOREDUMP nie jest zdefiniowana przez XSI. Jeśli jest dostępna, poprzedni przykład może być rozszerzony następująco:

```
/* testuj, czy proces potomny odebrał sygnał */
if (WIFSIGNALED(status))
{
    sig_no = WTERMSIG(status);
    printf("Signal number %d terminated child %d\n", sig_no, pid);
    if(WCOREDUMP(status))
        printf("...core dump created\n");
}
```

Format pliku rdzenia jest znany programom uruchomieniowym Unixa i można ich używać do badania stanu procesu w momencie zrzucania przez niego rdzenia. Może to być nadzwyczaj użyteczne, ponieważ program uruchomieniowy pozwala wskazać miejsce wystąpienia problemu.

Warto także wspomnieć o wywoływanej w prosty sposób procedurze abort:
abort();

Procedura abort wysyła do procesu wywołującego sygnał SIGABRT, powodując anormalne zakończenie, powiązane ze zrzutem rdzenia. Jest ona użyteczna jako środek pomagający w uruchamianiu programu, ponieważ pozwala procesowi na rejestrację jego bieżącego stanu w chwili, gdy dzieje się coś niewłaściwego. Ilustruje to także fakt, że proces może wysłać sygnał do samego siebie.

6.2 Obsługa sygnałów

Po otrzymaniu sygnału proces ma do wyboru jeden z trzech sposobów działania:

1. Podejmij działania domyślne. Domyślnym działaniem zwykle jest zakończenie procesu. Jednak dla sygnałów SIGUSR1 i SIGUSR2 polega ono na ignorowaniu sygnału. Natomiast dla SIGSTOP jest to zatrzymanie procesu (jego zawieszenie).
2. Zignoruj sygnał całkowicie i kontynuuj przetwarzanie. W dużych programach niespodziewane sygnały mogą powodować problemy. Na przykład nie pozwalać na zatrzymanie programu przez przypadkowe naciśnięcie klawisza przerwania, podczas gdy wykonuje on ważną aktualizację bazy danych.
3. Podejmij określone działania zdefiniowane przez użytkownika. Ilekroć program wychodzi, niezależnie od przyczyny, programista może chcieć wykonać gruntowne działania czyszczące, takie jak usuwanie plików roboczych.

W starszych wersjach Uniksa obsługa sygnałów była względnie prosta, choć czasem zawodziła. Nowsze procedury, które omawiamy, są ostrzejsze, ale bardziej złożone. Przed podaniem pierwszych przykładów konieczne jest jeszcze kilka objaśnień. Zaczniemy od **zestawów sygnałów** (ang. *signal sets*).

6.2.1 Zestawy sygnałów

Zestawy sygnałów są jednym z głównych parametrów przekazywanych do funkcji systemowych, które dotyczą sygnałów. Określają one po prostu listę sygnałów, z którymi chcesz coś zrobić.

Zestawy sygnałów są definiowane za pomocą typu `sigset_t`, zdefiniowanego z kolei w pliku nagłówkowym `<signal.h>`. Ten typ jest wystarczająco pojemy, aby zapamiętać reprezentację wszystkich zdefiniowanych w systemie sygnałów. Możesz teraz wybrać sposób wyrażenia zainteresowania określonymi sygnałami – zacząć od pełnego zestawu sygnałów i usuwać te, których nie chcesz, lub zacząć od pustego zestawu sygnałów i włączać te, których potrzebujesz. Kroki inicjacyjne są wykonywane za pomocą procedur `sigemptyset` i `sigfillset`. Dane zestawy sygnałów mogą być korygowane za pomocą funkcji `sigaddset` lub `sigdelset`, która odpowiednio dodaje lub usuwa sygnały.

Użycie

```
#include <signal.h>
/* funkcje inicjujące */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
/* funkcje manipulujące */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

Funkcje `sigemptyset` i `sigfillset` pobierają jeden parametr, którym jest wskaźnik do zmiennej typu `sigset_t`. Wywołanie funkcji `sigemptyset` inicjuje zestaw tak, że wszystkie sygnały są wyłączone. Funkcja `sigfillset` na odwrót, inicjuje parametr wskazywany przez `set` tak, że wszystkie sygnały są włączone. Aplikacje powinny wywołać `sigemptyset` albo `sigfillset` co najmniej raz dla każdej zmiennej typu `sigset_t`.

Funkcje `sigaddset` i `sigdelset` pobierają wskaźnik do zainicjowanego zestawu sygnałów i numer sygnału, który ma być odpowiednio dodany lub usunięty. Drugi parametr `signo` może być nazwą stałej symbolicznej sygnału, taką jak `SIGINT` lub, mniej przenośnie, rzeczywistym numerem sygnału.

W następującym przykładzie tworzymy dwa zestawy sygnałów, z których pierwszy jest początkowo pusty i dodajemy do niego sygnały `SIGINT` i `SIGQUIT`. Drugi zestaw jest początkowo pełny i usuwamy z niego sygnał `SIGCHLD`.

```
#include <signal.h>

sigset_t mask1, mask2;
.

.

/* utwórz pusty zestaw */
sigemptyset(&mask1);

/* dodaj sygnał */
sigaddset(&mask1, SIGINT);
sigaddset(&mask1, SIGQUIT);

/* utwórz pełny zestaw */
sigfillset(&mask2);

/* usuń sygnał */
sigdelset(&mask2, SIGCHLD);
.
```

6.2.2 Ustalanie działania sygnału: funkcja `sigaction`

Gdy już zdefiniowałeś zestaw sygnałów, możesz wybrać konkretną metodę obsługi sygnałów używając funkcji `sigaction`.

Użycie

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act,
              struct sigaction *oact);
```

Jak zobaczymy za chwilę, struktura `sigaction` zawiera zestaw sygnałów. Pierwszy parametr, `signo`, identyfikuje indywidualny sygnał, dla którego chcemy określić działanie. Aby odnieść jakikolwiek skutek, funkcja `sigaction` musi być wywołana przed odebraniem sygnału typu `signo`. Parametr `signo` może być ustawiony na którąś ze zdefiniowanych uprzednio nazw sygnałów, z wyjątkiem `SIGSTOP` i `SIGKILL`, które są dostarczane wyłącznie do – odpowiednio – zatrzymania (to jest zawieszenia) albo zakończenia procesu i nie mogą być obsługiwane w żaden inny sposób.

Drugi parametr, `act`, określa działanie, jakie chcesz ustawić dla `signo`. W istocie, trzeci parametr `oact` będzie po prostu wypełniony aktualnym ustawieniem albo też może być ustawiony na `NULL`. Jest on zdefiniowany w strukturze `sigaction`.

```
struct sigaction {
    void(*sa_handler)(int); /* działanie, które ma być podjęte */
    sigset_t sa_mask;      /* dodatkowe sygnały blokowane
                           podczas obsługi sygnału */
    int sa_flags;           /* znaczniki, które wpływają na
                           właściwości sygnału */
    void(*sa_sigaction)(int, siginfo_t *, void *); /* wskaźnik do programu obsługi
                           sygnału */
};
```

Struktura wygląda na bardzo złożoną, ale będziemy analizować ją krok po kroku. Pierwsze pole, `sa_handler`, identyfikuje działanie, jakie ma być podjęte po otrzymaniu sygnału `signo`. Może ono mieć jedną z trzech wartości:

1. `SIG_DFL` Specjalna nazwa symboliczna, która przywraca domyślne działanie systemu (zwykle kończy proces).
2. `SIG_IGN` Inna nazwa symboliczna, oznaczająca po prostu „ignoruj ten sygnał”. W przyszłości proces właśnie to zrobi. Nie może być używana dla sygnałów `SIGSTOP` i `SIGKILL`.
3. Adres funkcji, która pobiera argument całkowity. Jeśli jest ona zadeklarowana przed wywołaniem `sigaction`, `sa_handler` może być po prostu ustawiony na nazwę funkcji. Kompilator zakłada, że znasz jej adres. Funkcja będzie wykonana, kiedy zostanie otrzymany sygnał typu `signo`, a sama wartość `signo` zostanie wtedy przekazana do funkcji. Sterowanie będzie przekazane do funkcji, gdy tylko proces odbierze sygnał, niezależnie której części programu wykonuje. Kiedy funkcja powróci, sterowanie zostanie przekazane z powrotem do miejsca, w którym proces został przerwany. Ten mechanizm wyjaśnimy w naszym następnym przykładzie.

Drugie pole, `sa_mask`, demonstruje nasze pierwsze praktyczne zastosowanie zestawu sygnałów. Sygnały określone w `sa_mask` są blokowane w czasie spędzonym w funkcji określonej przez `sa_handler`. Nie znaczy to bynajmniej, że będą one ignorowane; są zatrzymane, aż zakończy się funkcja obsługi. Kiedy proces wprowadza funkcję, przechwycony sygnał zostaje też dodany do bieżącej maski sygnałów. Przekształca to sygnały w bardziej (ale nie całkowicie) godny zaufania mechanizm komunikacji.

Do modyfikacji zachowania `signo` – pierwotnie określonego sygnału – może być użyte pole `sa_flags`. Na przykład przez ustawienie `sa_flags` na `SA_RESETHAND` można ponownie ustawić na `SIG_DFL` działanie sygnału po powrocie z programu obsługi. Jeśli `sa_flags` jest ustawiony na `SA_SIGINFO`, do programu obsługi sygnału zostaną przekazane dodatkowe informacje. W tym przypadku pole `sa_handler` jest nadmiarowe, a używane będzie końcowe pole `sa_sigaction`. Struktura `siginfo_t` przekazywana do tego programu obsługi zawiera dodatkową informację o sygnale – na przykład jego numer, process-id wysyłającego i rzeczywisty user-id wysyłającego sygnał procesu. *XSI* określa, że dla naprawdę przenośnych programów proces powinien używać pola `sa_handler` albo `sa_sigaction`, ale nigdy obydwoch.

Poniższe przykłady dowodzą, że choć wszystko to wydaje się trudne, w rzeczywistości nie jest takie straszne.

Przykład 1: przechwytywanie sygnału SIGINT

Ten przykład pokazuje nie tylko, jak można przechwycić sygnał, lecz także rzuca więcej światła na podstawowy mechanizm sygnałów. Koncentruje się on na programie `sigex`, który po prostuiąże funkcję o nazwie `catchint` z sygnałem `SIGINT`, a następnie wykonuje szereg instrukcji `sleep` i `printf`. Zwróć uwagę, że definiujemy strukturę `sigaction` jako `static`. Wymusza to inicjację struktury, a w szczególności pola `sa_flags`, na zero.

```
/* sigex -- pokazuje działanie sigaction */
#include <signal.h>

main()
{
    static struct sigaction act;

    /* deklaracja catchint, później używanego jako program obsługi */
    void catchint(int);

    /* ustaw działanie podejmowane po odebraniu SIGINT */
    act.sa_handler = catchint;

    /* utwórz pełną maskę */
    sigfillset(&(act.sa_mask));

    /* przed wywołaniem sigaction, SIGINT powinien
     * zakończyć proces (działanie domyślne) */
    sigaction(SIGINT, &act, NULL);

    /* po otrzymaniu SIGINT sterowanie będzie przekazane
     * do catchint */

    printf("sleep call #1\n");
    sleep(1);
    printf("sleep call #2\n");
    sleep(1);
```

```

printf("sleep call #3\n");
sleep(1);
printf("sleep call #4\n");
sleep(1);

printf("Exiting \n");
exit (0);
}

/* trywialna funkcja do obsługi SIGINT */
void catchint(int signo)
{
    printf("\nCATCHINT: signo=%d\n", signo);
    printf("CATCHINT: returning\n\n");
}

```

Pozostawiony samemu sobie, program `sigex` daje następujący wynik:

```

$ sigex
sleep call #1
sleep call #2
sleep call #3
sleep call #4
Exiting

```

Użytkownik może jednak przerwać działanie `sigex` przez naciśnięcie klawisza przerwania. Jeśli zrobi to, zanim `sigex` miał szansę wykonać funkcję `sigaction`, proces po prostu zakończy się. Jeśli natomiast naciśnie klawisz po wywołaniu `sigaction`, sterowanie zostanie przekazane do funkcji `catchint`, jak poniżej:

```

$ sigex
sleep call #1
<interrupt>          (użytkownik nacisnął klawisz przerwania)

```

CATCHINT: signo = 2

CATCHINT: returning

```

sleep call #2
sleep call #3
sleep call #4
Exiting

```

Zauważ, jak przekazywane jest sterowanie z głównego ciała programu do `catchint`. Kiedy `catchint` kończy się, sterowanie powraca do miejsca, w którym program został przerwany. Równie łatwo program `sigex` może być przerwany w innym miejscu:

```

$ sigex
sleep call #1
sleep call #2
<interrupt>          (użytkownik nacisnął klawisz przerwania)

```

CATCHINT: signo = 2

CATCHINT: returning

```

sleep call #3
sleep call #4
Exiting

```

Przykład 2: ignorowanie sygnału SIGINT

Przypuśćmy, że chcemy, aby proces ignorował sygnał przerwania `SIGINT`. Wszystko, co potrzebujemy robić, to zastąpić następującą linię programu:

```
act.sa_handler = catchint;
```

przez:

```
act.sa_handler = SIG_IGN;
```

Po wykonaniu tej instrukcji klawisz przerwania nie będzie dawał żadnego efektu. Może on być ponownie włączony za pomocą:

```
act.sa_handler = SIG_DFL;
sigaction(SIGINT, &act, NULL);
```

Mogliwe jest jednocześnie ignorowanie kilku sygnałów. Na przykład

```
act.sa_handler = SIG_IGN;
sigaction(SIGINT, &act, NULL);
sigaction(SIGQUIT, &act, NULL);
```

blokuje zarówno `SIGINT`, jak i `SIGQUIT`. Jest to przydatne dla programów, które nie powinny być przerywane za pomocą klawiatury.

Pewne powłoki używają tej techniki do zapewnienia, że proces działający w tle nie jest zatrzymywany, gdy użytkownik wciśnie klawisz przerwania. Dzieje się tak, ponieważ sygnały, które są ignorowane przez proces, są ignorowane jeszcze po wywołaniu `exec`. Dlatego powłoka może wywołać `sigaction`, aby upewnić się, że `SIGQUIT` i `SIGINT` są ignorowane, a następnie za pomocą `exec` uruchomić nowy program.

Przykład 3: przywrócenie poprzedniego działania

Jak widzieliśmy powyżej, `sigaction` może mieć wypełniony swój trzeci parametr, `oact`. Pozwala to nam zachować i przywrócić poprzedni stan sygnału, co pokazuje następny przykład:

```

#include <signal.h>

static struct sigaction act, oact;

/* zachowaj stare działanie SIGTERM */
sigaction(SIGTERM, NULL, &oact);

/* ustaw nowe działanie SIGTERM */
act.sa_handler = SIG_IGN;
sigaction(SIGTERM, &act, NULL);

/* tu wykonaj całą pracę ... */

/* teraz przywróć stare działanie */
sigaction(SIGTERM, &oact, NULL);

```

Przykład 4: eleganckie wyjście

Załóżmy, że program używa tymczasowego pliku roboczego. Następująca prosta procedura usuwa ten plik:

```
/* eleganckie wyjście z programu */
#include <stdio.h>
#include <stdlib.h>

void g_exit(int s)
{
    unlink("tempfile");
    fprintf(stderr, "Interrupted -- exiting\n");
    exit(1);
}
```

Może być ona połączona z konkretnym sygnałem, jak poniżej:

```
extern void g_exit(int);
.

.

static struct sigaction act;
act.sa_handler = g_exit;
sigaction(SIGINT, &act, NULL);
```

Po tym wywołaniu, gdy użytkownik wciśnie klawisz przerwania, sterowanie zostanie automatycznie przekazane do `g_exit`. Zawartość `g_exit` może być rozszerzana w zależności od wymaganej liczby operacji czyszczących.

6.2.3 Sygnały i funkcje systemowe

W większości przypadków, jeśli do procesu jest wysyłany sygnał w chwili, gdy wykonuje on funkcję systemową, sygnał nie odnosi żadnego skutku, dopóki funkcja się nie zakończy. Jednak kilka funkcji systemowych zachowuje się inaczej i mogą być one przerwane przez sygnał. Dotyczy to funkcji `read`, `write` lub `open` w odniesieniu do powolnych urządzeń (takich jak terminal, ale już nie plik dyskowy), oraz funkcji `wait` lub `pause` (które omówimy w odpowiednim czasie). We wszystkich przypadkach, jeśli proces przechwyci wywołanie, przerwana funkcja systemowa zwraca `-1` i umieszcza `EINTR` w `errno`. Ten rodzaj sytuacji może być obsłużony za pomocą następującego kodu:

```
if(write(tfd, buf, size) < 0)
{
    if(errno == EINTR)
    {
        warn("Write interrupted");
        .
        .
    }
}
```

W tym przypadku, jeśli program chce ponownie wywołać funkcję systemową `write`, musi użyć pętli i instrukcji `continue`. Jednak jeśli funkcja `sigaction` została przerwana w ten sposób, pozwala na automatyczne ponowne uruchomienie

nie funkcji systemowej. Można to osiągnąć za pomocą ustawienia zmiennej `sa_flags` w strukturze `sigaction` na `SA_RESTART`. Jeśli ten znacznik jest ustawiony, funkcja systemowa zostanie ponownie uruchomiona i zmieniona błąd `errno` nie będzie ustawiona.

Jeszcze jedna ważna uwaga: sygnały Unixa zwykle nie mogą być odkładane na stosie. Mówiąc ściślej, dla danego procesu w danej chwili nie może istnieć więcej niż jeden zaległy sygnał danego typu, chociaż może występować więcej niż jeden zaległy typ sygnałów. Fakt, że sygnały nie mogą być odkładane na stosie, oznacza, że nie powinny być używane jako w pełni godna zaufania metoda komunikacji międzyprocesowej, ponieważ proces nigdy nie jest pewny, czy sygnał przez niego wysłany nie został utracony.

Ćwiczenie 6.1 Zmień `smallsh` z ostatniego rozdziału w ten sposób, żeby obsługiwał przerwania jak prawdziwa powłoka. Upewnij się, że procesy działające w tle nie są zatrzymywane przez sygnał `SIGINT` lub `SIGHUP`. Kilka wariantów powłoki (mianowicie *powłoka C* i *powłoka Korn*) obsługuje proces w tle przez umieszczanie go w innej grupie procesów. Jakie są zalety i wady takiego podejścia? (W ostatniej pracy POSIX została wprowadzona możliwość odkładania sygnałów na stosie – ale jako opcja).

6.2.4 Podprogramy `sigsetjmp` i `siglongjmp`

Czasami w momencie otrzymania sygnału istnieje potrzeba przeskoczenia do poprzedniej pozycji w programie. Na przykład możesz pozwolić użytkownikowi powrócić do głównego menu programu w chwili, gdy naciśnie on klawisz przerwania. Można to zrobić za pomocą dwóch specjalnych podprogramów o nazwach `sigsetjmp` i `siglongjmp`. (Istnieją alternatywne podprogramy o nazwach `setjmp` i `longjmp`, z różną obsługą sygnałów). Podprogram `sigsetjmp` zachowuje bieżącą pozycję programu i maskę sygnału przez zapamiętanie środowiska stosu, a `siglongjmp` przekazuje sterowanie wstecz, do zachowanej pozycji. W pewnym sensie `siglongjmp` jest rodzajem dalekiego, a nie lokalnego skoku. Należy zdać sobie sprawę, że `siglongjmp` nigdy nie wraca, ponieważ ramka stosu jest skracana do miejsca, gdzie została zapamiętana pozycja. Jak zobaczymy, istnieje odpowiedni podprogram `sigsetjmp`, który powraca.

Użycie

```
#include <setjmp.h>
/* zachowuje pozycję programu */
int sigsetjmp(sigjmp_buf env, int savemask);
/* powraca do zachowanej pozycji */
void siglongjmp(sigjmp_buf env, int val);
```

Pozycja programu jest zachowywana w obiekcie typu `sigjmp_buf`, zdefiniowanym w standardowym pliku nagłówkowym `<setjmp.h>`. Jeśli w wywołaniu podprogramu `sigsetjmp` wartość `savemask` jest niezerowa, czyli TRUE, wtedy `sigsetjmp` zachowuje bieżącą maskę sygnalów (to znaczy stan i działania związane ze wszystkimi sygnalami) oraz środowisko, mogą więc być one przywrócone, gdy `siglongjmp` przekaże sterowanie wstecz, do tej zachowanej pozycji. Powrót `sigsetjmp` jest znaczący: jeśli podprogram `sigsetjmp` został wywołany z `siglongjmp`, zwróci wartość różną od 0, w rzeczywistości `val` z `siglongjmp`. Jeśli jest jednak wywoływany jako następna sekwencyjna instrukcja, zwróci 0.

Następujący przykład wyjaśnia te zagadnienia:

```
/* przykład użycia sigsetjmp i siglongjmp */

#include <sys/types.h>
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>

sigjmp_buf position;

main()
{
    static struct sigaction act;
    void goback(void);

    /* zachowaj bieżącą pozycję */
    if(sigsetjmp(position, 1) == 0)
    {
        act.sa_handler = goback;
        sigaction(SIGINT, &act, NULL);
    }

    domenu();
}

void goback(void)
{
    fprintf(stderr, "\nInterrupted\n");

    /* powróć do zachowanej pozycji */
    siglongjmp(position, 1);
}
```

Jeśli użytkownik naciśnie przerwanie po wywołaniu `sigaction`, sterowanie będzie przekazywane najpierw do `goback`. To z kolei wywoła `siglongjmp` i sterowanie zostanie przekazane w tył do miejsca, gdzie `sigsetjmp` zarejestrował pozycję pro-

gramu. Wykonanie programu trwa tak, jakby odpowiednie wywołanie `sigsetjmp` właśnie wróciło. Wartość zwracana przez `sigsetjmp` jest w tym przypadku pobierana z drugiego parametru `siglongjmp`.

6.3 Blokowanie sygnałów

Jeśli program wykonuje odpowiedzialne zadanie, np. aktualizację bazy danych, byłoby dobrze zabezpieczyć go przed przerwaniami w kluczowych sytuacjach. Zamiast ignorowania wszystkich nadchodzących sygnałów, jak widzieliśmy powyżej, proces może blokować sygnały, co oznacza, że nie będą one obsługiwane, dopóki proces nie zakończy swojej delikatnej operacji.

Funkcją systemową, pozwalającą procesowi zablokować konkretny sygnał, jest `sigprocmask`, zdefiniowana następująco:

Użycie

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

Parametr `how` mówi funkcji `sigprocmask`, jakie działanie ma przeciwwziąć. Na przykład może to być `SIG_SETMASK`, co oznacza blokowanie sygnałów ustawionych w drugim parametrze `set`. Trzeci parametr jest po prostu wypełniany bieżącą maską blokowanych sygnałów – jeśli nie potrzebujesz jej znać, ustaw go na `NULL`. Oto przykład, który to wyjaśni:

```
sigset_t set1;
.

.

/* wypełnij całkowicie zestaw sygnałów */
sigfillset(&set1);

/* ustaw blokadę */
sigprocmask(SIG_SETMASK, &set1, NULL);

/* wykonaj wyjątkowo krytyczny kod ... */

/* usuń blokadę sygnałów */
sigprocmask(SIG_UNBLOCK, &set1, NULL);
```

Zwróć uwagę na użycie `SIG_UNBLOCK` do usunięcia blokady sygnałów. Zauważ też, że podając dla pierwszego parametru wartość `SIG_BLOCK` zamiast `SIG_SETMASK`, dodamy sygnały określone w `set` do bieżącego zestawu sygnałów.

Ten bardziej złożony przykład pokazuje blokowanie wszystkich sygnałów w bardzo krytycznym fragmencie kodu, a następnie tylko `SIGINT` i `SIGQUIT` w mniej krytycznym fragmencie.

```
/* blokowanie sygnałów -- pokazuje użycie sigprocmask */
#include <signal.h>

main()
{
    sigset(SIG_BLOCK, set1, set2);

    /* wypełnij całkowicie zestaw sygnałów */
    sigfillset(&set1);

    /* utwórz sygnał, który nie jest
     * SIGINT ani SIGQUIT */
    sigfillset(&set2);
    sigdelset(&set2, SIGINT);
    sigdelset(&set2, SIGQUIT);

    /* wykonaj nie krytyczny kod ... */

    /* ustaw blokadę */
    sigprocmask(SIG_SETMASK, &set1, NULL);

    /* wykonaj wyjątkowo krytyczny kod ... */

    /* ustaw mniejszą blokadę */
    sigprocmask(SIG_UNBLOCK, &set2, NULL);

    /* wykonaj mniej krytyczny kod ... */

    /* usuń wszystkie blokady sygnałów */
    sigprocmask(SIG_UNBLOCK, &set1, NULL);
}
```

Ćwiczenie 6.2 Zmodyfikuj procedurę `g_exit` wprowadzoną w przykładzie 4 w podrozdziale 6.2.2 tak, aby ignorowała `SIGINT` i `SIGQUIT` w czasie trwania funkcji.

6.4 Wysyłanie sygnałów

6.4.1 Wysyłanie sygnałów do innych procesów: funkcja `kill`

Funkcja systemowa `sigaction` obsługuje sygnały wysyłane przez inne procesy. Operacja odwrotna – bieżącego wysyłania sygnału – wykonywana jest przez funkcję o dramatycznej nazwie `kill` (zabij). Użycie funkcji `kill` jest następujące:

Użycie

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

int kill (pid_t pid, int sig);
```

Pierwszy parametr `pid` określa proces albo procesy, do których sygnał `sig` będzie wysłany. Zwykle `pid` to dodatnia liczba (w tym przypadku będzie to faktyczny identyfikator procesu (process-id)). Tak więc instrukcja:

```
kill(7421, SIGTERM);
```

oznacza *wyslij sygnał SIGTERM do procesu z process-id równym 7421*. Ponieważ proces, który wywołuje funkcja `kill` musi znać identyfikator procesu, do którego wysyła sygnał, funkcji `kill`, używa się najczęściej między powiązanymi procesami, na przykład procesem rodzicielskim i potomnym. Warto też zauważyć, że proces może wysłać sygnał do samego siebie.

Istnieje kilka zagadnień, dotyczących przywilejów. Aby wysłać sygnał do procesu, rzeczywisty albo efektywny user-id procesu wysyłającego musi być zgodny z rzeczywistym albo efektywnym user-id odbiorcy. Jak zwykle proces super-użytkownika może wysłać sygnał do każdego innego procesu. Jeśli proces o niższym poziomie przywilejów (nie będący procesem super-użytkownika) próbuje wysłać sygnał do procesu, który należy do innego użytkownika, wtedy funkcja `kill` zawodzi, zwraca -1 i umieszcza wartość EPERM w `errno`. (Innymi możliwymi wartościami dla `errno` po wywołaniu `kill` są ESRCH, oznaczająca, że nie ma takiego procesu, albo EINVAL, jeśli `sig` nie jest ważnym numerem sygnału).

Parametr `pid` funkcji `kill` może przybierać inne wartości, które mają specjalne znaczenie:

- Jeśli `pid` jest równy zeru, sygnał będzie wysłany do wszystkich procesów, które należą do tej samej grupy procesów, co nadawca. Dotyczy to również nadawcy.
- Jeśli `pid` jest równy -1, a efektywny user-id procesu nie jest super-użytkownikiem, wtedy sygnał zostanie wysłany do wszystkich procesów z rzeczywistym user-id równym efektywnemu user-id nadawcy. Ponownie, dotyczy to również nadawcy.
- Jeśli `pid` jest równy -1, a efektywny user-id procesu to super-użytkownik, sygnał jest wysyłany do wszystkich procesów z wyjątkiem pewnych specjalnych procesów systemowych (ten ostatni zakaz ma zastosowanie do wszystkich prób wysłania sygnału do grupy procesów, ale tutaj jest najważniejszy).
- Wreszcie, jeśli `pid` jest mniejszy niż zero, ale różny od -1, sygnał będzie wysłany do wszystkich procesów z group-id procesu równym bezwzględnej wartości `pid`. Dotyczy to również nadawcy, jeśli jego group-id procesu spełnia ten warunek.

A oto przykład: `synchro` tworzy dwa procesy. Oba zapisują kolejno komunikaty do standardowego wyjścia. Synchronizują się za pomocą wzajemnego wysyłania sygnałów `SIGUSR1`.

```
/* synchro -- przykład funkcji kill */

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

```

int ntimes = 0;

main ()
{
    pid_t pid, ppid;
    void p_action(int), c_action(int);
    static struct sigaction pact, cact;

    /* ustaw działanie SIGUSR1 dla rodzica */
    pact.sa_handler = p_action;
    sigaction(SIGUSR1, &pact, NULL);

    switch(pid = fork()) {
    case -1:           /* błąd */
        perror("synchro");
        exit(1);
    case 0:            /* potomek */

        /* ustaw działanie dla potomka */
        cact.sa_handler = c_action;
        sigaction(SIGUSR1, &cact, NULL);

        /* pobierz process-id rodzica */
        ppid = getppid();

        for(;;)
        {
            sleep(1);
            kill(ppid, SIGUSR1);
            pause();
        }
        /* pętla bez końca */

    default:           /* rodzic */
        for(;;)
        {
            pause();
            sleep(1);
            kill(pid, SIGUSR1);
        }
        /* nigdy się nie zakończy */
    }
}

void p_action(int sig)
{
    printf("Parent caught signal #%d\n", ++ntimes);
}

void c_action(int sig)
{
    printf("Child caught signal #%d\n", ++ntimes);
}

```

Każdy proces znajduje się w pętli, czekając w zawieszeniu, aż odbierze sygnał od drugiego. Jest to wykonywane za pomocą funkcji systemowej pause, która po prostu zawiesza wykonywanie, dopóki nie przybędzie sygnał (zobacz w podrozdziale 6.4.3). Każdy proces drukuje wtedy komunikat i wysyla sygnał za pomocą kill. Proces potomny rozpoczyna ten mecz (zwróć uwagę na kolejność instrukcji w każdej pętli). Oba procesy są kończone, kiedy użytkownik naciśnie klawisz przerwania. Przykładowy dialog może wyglądać tak:

```

$ synchro
Parent caught signal #1
Child caught signal #1
Parent caught signal #2
Child caught signal #2
<interrupt>
$
```

(użytkownik naciągnął klawisz przerwania)

6.4.2 Wysyłanie sygnałów do samego siebie: funkcje raise i alarm

Funkcja raise po prostu wysyła sygnał do procesu wykonującego wywołanie.

Użycie

```
#include <signal.h>
int raise(int sig);
```

Parametr sig jest wysyłany do procesu wywołującego, a w przypadku powodzenia funkcja raise zwraca 0.

Funkcja alarm to prosta i przydatna funkcja, która ustawia zegar alarmu (ang. *alarm clock*) procesu. Aby powiedzieć procesowi, że czas minął, używa się sygnałów.

Użycie

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

Parametr secs podaje tu czas w sekundach do alarmu. Kiedy ten czas upłynie, do procesu zostanie wysłany sygnał SIGALRM. Wywołanie:

```
alarm(60);
```

spowoduje więc po 60 sekundach wysłanie sygnału SIGALRM. Zwróć uwagę, że funkcja alarm różni się od funkcji sleep, która zawiesza wykonanie procesu; funkcja alarm wraca natychmiast i proces kontynuuje wykonanie w normalny sposób, przynajmniej do otrzymania sygnału SIGALRM. W rzeczywistości aktywny zegar alarmu powinien także przedostawać się przez wywołanie exec. Jednak po wywołaniu funkcji fork zegar alarmu w procesie potomnym jest wyłączany.

Alarm może być wyłączony za pomocą wywołania funkcji alarm z parametrem zerowym:

```
/* wyłącz zegar alarmu */
alarm(0);
```

Wywołania funkcji alarm nie są odkładane na stosie: inaczej mówiąc, jeśli wywołasz dwukrotnie funkcję alarm, drugie wywołanie zastąpi pierwsze. Jednak wartość zwracana przez alarm podaje czas pozostający dla poprzedniego zegara alarmu, który może być w razie potrzeby zapamiętany.

Funkcja alarm staje się użyteczna, gdy programista potrzebuje określić limit czasu dla pewnej działalności. Podstawowa idea jest prosta: wywołana zostaje funkcja alarm i proces kontynuuje zadanie. Jeśli zadanie będzie zakończone w krótszym czasie, zegar alarmu zostanie wyłączony. Jeśli natomiast potrwa zbyt długo, proces zostanie przerwany za pomocą SIGALRM i podejmie działania korygujące.

Następująca funkcja quickreply używa tego podejścia do wymuszenia odpowiedzi od użytkownika. Pobiera ona jeden argument, znak zachęty, i zwraca wskaźnik do napisu zawierającego linię wejściową lub pusty wskaźnik, jeśli nic nie wpisano po pięciu powtórzeniach. Zwróć uwagę, że za każdym razem, gdy quickreply przypomina się użytkownikowi, wysyła [Ctrl+G] do terminala. Powoduje to sygnał dźwiękowy na większości terminali sprzętowych lub emulatorów.

Funkcja quickreply wywołuje procedurę gets, która pochodzi ze Standardowej Biblioteki I/O. Procedura gets umieszcza następną linię ze standardowego wejścia w tablicy char. Zwraca ona wskaźnik do tablicy lub pusty wskaźnik w przypadku końca pliku albo błędu. Zauważ, że SIGALRM zostaje przechwycony przez procedurę przerwania catch. Jest to ważne, ponieważ domyślne działanie związane z SIGALRM to oczywiście zakończenie. Procedura catch ustawia znacznik o nazwie timed_out. Funkcja quickreply sprawdza go, aby zbadać, czy istotnie przekroczyła czas oczekiwania.

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

#define TIMEOUT      5      /* w sekundach */
#define MAXTRIES    5
#define LINESIZE     100
#define CTRL_G       '\007' /* sygnał dźwiękowy ASCII */
#define TRUE         1
#define FALSE        0

/* sprawdza, czy minał czas oczekiwania */
static int timed_out;

/* zawiera linię wejściową */
static char answer[LINESIZE];

char *quickreply(char *prompt)
{
    void catch(int);
    int ntries;
    static struct sigaction act, oact;
```

```
/* przechwyć SIGALRM + zachowaj poprzednie działanie */
act.sa_handler = catch;
sigaction(SIGALRM, &act, &oact);

for(ntries=0; ntries<MAXTRIES; ntries++)
{
    timed_out = FALSE;
    printf("\n%s > ", prompt);

    /* ustaw zegar alarmu */
    alarm(TIMEOUT);

    /* pobierz linię wejściową */
    gets(answer);

    /* wyłącz alarm */
    alarm(0);
    /* jeśli czas oczekiwania TRUE, nie ma odpowiedzi */
    if(!timed_out)
        break;
}

/* przywróć stare działanie */
sigaction(SIGALRM, &oact, NULL);

/* zwróć odpowiednią wartość */
return(ntries == MAXTRIES ? ((char *)0) : answer);
}

/* wykonaj, gdy SIGALRM odebrany */
void catch(int sig)
{
    /* ustaw znacznik timed_out */
    timed_out = TRUE;

    /* sygnał dźwiękowy */
    putchar(CTRL_G);
}
```

6.4.3 Funkcja systemowa pause

Dla towarzystwa funkcji alarm, Unix dostarcza funkcję systemową pause, którą jest bardzo łatwo wywołać:

Użycie

```
#include <unistd.h>
int pause(void);
```

Funkcja pause zawiesza wywołujący proces (w ten sposób, że nie będzie marnował czasu CPU), aż do otrzymania dowolnego sygnału, np. SIGALRM. Jeśli sygnał

powoduje normalne zakończenie, wtedy zdarzy się tylko to. Jeśli sygnał jest ignorowany przez proces, pause ignoruje go również. Jeśli sygnał jest jednak przechwytywany, wtedy gdy kończy się stosowna procedura przerwania, funkcja pause zwraca -1 i umieszcza EINTR w errno (dlaczego?).

Program tml (skrót od „powiedz mi później” – ang. *tell me later*) używa kolejno funkcji alarm i pause, aby wyświetlić komunikat za określoną liczbę minut. Jest on wywoływany następująco:

```
$ tml # minuty tekst-komunikatu
```

Na przykład:

```
$ tml 10 koniec pracy
```

W celu uzyskania większego efektu komunikat jest poprzedzany trzema znakami [Ctrl+G] lub dzwonkami. Zwróć uwagę, jak tml tworzy wykonujący całą pracę proces działający w tle, pozwalając użytkownikowi wykonywać inne zadania.

```
/* tml -- program tell-me-later */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

#define TRUE      1
#define FALSE     0
#define BELLS    "\007\007\007" /* sygnał dźwiękowy ASCII */

int alarm_flag = FALSE;

/* procedura obsługi SIGALRM */
void setflag(int sig)
{
    alarm_flag = TRUE;
}

main(int argc, char **argv)
{
    int nsecs, j;
    pid_t pid;
    static struct sigaction act;

    if( argc<=2 )
    {
        fprintf(stderr, "Usage: tml #minutes message\n");
        exit(1);
    }

    if((nsecs=atoi(argv[1])*60) <= 0)
    {
        fprintf(stderr, "tml: invalid time\n");
        exit(2);
    }
}
```

```
)
/* fork tworzy proces w tle */
switch (pid = fork()){
    case -1:                                /* błąd */
        perror ("tml");
        exit (1);
    case 0:                                   /* potomek */
        break;
    default:                                 /* rodzic */
        printf("tml process-id %d\n", pid);
        exit (0);
}

/* ustaw działanie alarmu */
act.sa_handler = setflag;
sigaction(SIGALRM, &act, NULL);

/* włącz zegar alarmu */
alarm(nsecs);

/* pauza do sygnału ... */
pause ();

/* jeśli SIGALRM, drukuj komunikat */
if(alarm_flag == TRUE)
{
    printf (BELLS);
    for(j = 2; j < argc; j++)
        printf("%s ", argv[j]);
    printf ("\n");
}

exit (0);
}
```

Na podstawie tego przykładu powinieneś wyrobić sobie pogląd na to, jak działa podprogram sleep za pomocą wywołania najpierw funkcji alarm, a później pause.

Ćwiczenie 6.3 Napisz własną wersję funkcji sleep. Upewnij się, że zachowuje ona poprzedni stan zegara alarmu i przywraca go przy wyjściu. (Sprawdź w podręczniku swojego systemu pełną specyfikację sleep).

Ćwiczenie 6.4 Zmodyfikuj program tml używając swojej wersji sleep.

ROZDZIAŁ 7

Komunikacja międzyprocesowa za pomocą potoków

7.1 Potoki

7.2 Pliki FIFO lub nazwane potoki

Jeśli nad wykonaniem zadania pracują dwa (albo więcej) procesy, wiadomo, że muszą one wspólnie korzystać z danych. Sygnały – użyteczne w sytuacjach niezwykłych lub błędnych – są całkowicie nieodpowiednie do przekazywania dużej ilości informacji z jednego procesu do drugiego. Jednym z możliwych sposobów rozwiązywania tego problemu dla procesów jest wspólne korzystanie z plików, ponieważ nic nie wzbrania kilku procesom odczytywać albo zapisywać ten sam plik równocześnie. Jednak może to być nieefektywne, a poza tym należy zwracać uwagę na unikanie problemów spornych.

Aby rozwiązać ten problem, Unix dostarcza konstrukcji nazywanej **potokiem** (ang. *pipe*) (jak również innych konstrukcji, które omówimy w następnych rozdziałach). Potok jest używany najczęściej jako jednokierunkowy kanał komunikacyjny, łączący ze sobą powiązane procesy, stanowiąc zarazem uogólnienie pojęcia pliku w Uniksie. Jak zobaczymy, proces może wysłać dane „w dół” potoku za pomocą funkcji systemowej `write`, a inny proces może odebrać dane na drugim końcu potoku, używając funkcji systemowej `read`.

7.1 Potoki

7.1.1 Potoki na poziomie poleceń

Większość użytkowników Uniksa styka się z potokami na poziomie poleceń. Na przykład polecenie

```
$ pr doc | lp
```

powoduje, że powłoka rozpoczyna jednocześnie polecenia `pr` i `lp`. Symbol `|` w wierszu polecenia oznacza, że powłoka tworzy potok łączący standardowe wyjście `pr` ze standardowym wejściem `lp`. Końcowym wynikiem tego polecenia powinna być wersja pliku `doc` z ładnie ponumerowanymi stronami, wysłana do drukarki.

Przeprowadźmy dalszą analizę polecenia. Program `pr` po lewej stronie symbolu potoku nie wie, że jego standardowe wyjście jest wysyłane do potoku. On tylko jak zwykle zapisuje do standardowego wyjścia, nie robiąc żadnych specjalnych przygotowań. Podobnie program `lp` po prawej stronie czyta z potoku, jakby jego standardowe wejście pochodziło z klawiatury lub zwykłego pliku dyskowego. Ostateczny efekt jest taki, jak wykonanie następującej sekwencji poleceń:

```
$ pr doc > tmpfile
$ lp < tmpfile
$ rm tmpfile
```

Przebieg sterowania wzduż potoku jest obsługiwany w sposób automatyczny i niewidoczny dla użytkownika. Tak więc, jeśli `pr` tworzy informację zbyt szybko, jego wykonanie zostaje zawieszone. Będzie ponownie uruchomiony, gdy `lp` nadrobí zaległości i ilość danych w potoku spadnie do dopuszczalnego poziomu.

Potoki są jedną z najmocniejszych i najbardziej charakterystycznych cech Uniksa, szczególnie na poziomie polecen. Pozwalają na połączenie dowolnej sekwencji polecen. Dlatego programy uniksowe mogą być opracowywane jako ogólne narzędzia, które odczytują ze swojego standardowego wejścia, zapisują do standardowego wyjścia i wykonują pojedyncze, dobrze zdefiniowane zadanie. Z tych podstawowych cegiełek można zbudować za pomocą potoków bardziej złożone wiersze polecen. Na przykład:

```
$ who | wc -l
```

Łączy potokiem wyjście `who` z wejściem programu liczenia słów `wc`. Opcja `-l` oznacza, że `wc` ma liczyć tylko linie. Dlatego wartość wyświetlana w końcu przez `wc` to liczba zarejestrowanych w systemie użytkowników.

7.1.2 Programowanie z potokami

Wewnątrz programu potok jest tworzony za pomocą funkcji systemowej o nazwie `pipe`. Jeśli wywołanie było pomyślne, zwraca dwa deskryptory plików: jeden do zapisu do potoku, a drugi do odczytu z niego. Funkcja `pipe` została zdefiniowana następująco:

Użycie

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Parametr `filedes` to tablica dwóch liczb całkowitych, zawierająca deskryptory plików, które identyfikują potok. Jeśli wywołanie jest pomyślne, `filedes[0]` zostaje otwarty dla odczytu z potoku, a `filedes[1]` otwarty dla zapisu do potoku.

Wywołanie `pipe` może zakończyć się i zwrócić `-1`. Zdarzy się to, gdyby spowodowało ono otwarcie większej liczby deskryptorów plików, niż limit procesu na jednego użytkownika (w tym przypadku `errno` zawiera `EMFILE`) lub gdyby tablica otwartych plików jądra uległa przepelnieniu (wtedy `errno` będzie zawierać `ENFILE`).

Po utworzeniu potok może być wykorzystywany w prosty sposób, za pomocą `read` i `write`. Następujący przykład tworzy potok, zapisuje do niego trzy komunikaty, a następnie odczytuje je z powrotem:

```
/* pierwszy przykład potoku */
#include <unistd.h>
#include <stdio.h>

/* ta wartość zawiera końcowe zero */
#define MSGSIZE 16

char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main()
{
    char inbuf[MSGSIZE];
    int p[2], j;

    /* otwórz potok */
    if(pipe(p) == -1)
    {
        perror("pipe call");
        exit(1);
    }

    /* zapisz do potoku */
    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);

    /* odczytaj z potoku */
    for(j = 0; j < 3; j++)
    {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
    }

    exit(0);
}
```

Wyjście tego programu wygląda następująco:

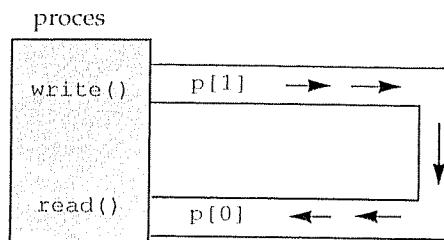
```
hello, world #1
hello, world #2
hello, world #3
```

Zwróć uwagę, że komunikaty są czytane w tej kolejności, w jakiej zostały zapisane. Potok traktuje dane na bazie *pierwszy na wejściu – pierwszy na wyjściu* (ang. *first-in first-out, FIFO*). Innymi słowy, to co umieściszt najpierw w potoku, zostanie odczytane jako pierwsze. Ta kolejność nie może być zmieniona, ponieważ `lseek` nie działa w potoku.

Robimy tak wprawdzie w przykładzie, ale proces nie musi czytać z potoku porcjami tej samej wielkości, jakie były zapisywane. Na przykład potok może być

zapisywany blokami 512-bajtowymi i odczytywany znak po znaku, zupełnie jak zwykły plik. Jednak, jak zobaczymy w podrozdziale 7.2, istnieją pewne zalety używania porcji o stałej wielkości.

Działanie przykładu pokazane jest na rysunku 7.1, który powinien nam unaoczyć, że proces wysyłania danych do siebie za pomocą potoku jest rodzajem mechanizmu pętli zwrotnej. Proces mówi tu tylko do siebie, a więc może się to wydawać bezcelowe.



Rysunek 7.1 Pierwszy przykład potoku

Prawdziwa wartość potoku staje się widoczna, gdy jest on używany w połączeniu z funkcją systemową fork, gdzie może być wykorzystany fakt, że deskryptory plików pozostają otwarte przy wywołaniu fork. Pokazuje to下一个 przykład. Tworzy on potok i wywołuje fork. Następnie proces potomny zapisuje szereg komunikatów do swojego rodzica.

```
/* drugi przykład potoku */
#include <unistd.h>
#include <stdio.h>

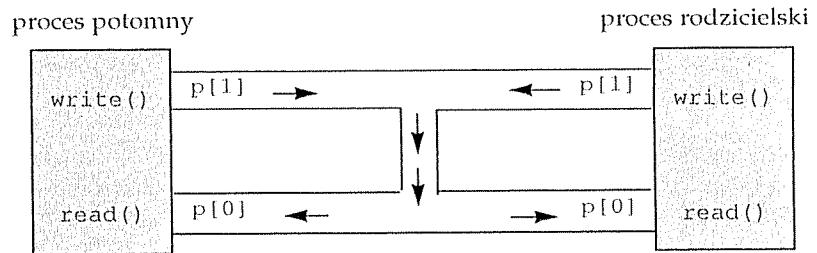
#define MSGSIZE 16

char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main()
{
    char inbuf[MSGSIZE];
    int p[2], j;
    pid_t pid;
    /* otwórz potok */
    if(pipe(p) == -1)
    {
        perror("pipe call");
        exit(1);
    }
    switch(pid = fork())
    {
    case -1:
        perror("fork call");
        exit(2);
    case 0:
        /* jeśli potomek, to zapisz do potoku */
        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);
        break;
    default:
        /* jeśli rodzic, to odczytaj z potoku */
        for(j = 0; j < 3; j++)
        {
            read(p[0], inbuf, MSGSIZE);
            printf("%s\n", inbuf);
        }
        wait(NULL);
    }
    exit(0);
}
```

```
write(p[1], msg1, MSGSIZE);
write(p[1], msg2, MSGSIZE);
write(p[1], msg3, MSGSIZE);
break;
default:
    /* jeśli rodzic, to odczytaj z potoku */
    for(j = 0; j < 3; j++)
    {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
    }
    wait(NULL);
}
exit(0);
}
```

Ten przykład jest przedstawiony w formie wykresu na rysunku 7.2. Pokazuje on potok łączący dwa procesy. Jak widać, proces rodzicielski i proces potomny mają otwarte dwa deskryptory pliku, pozwalające odczytywać i zapisywać do potoku. Każdy z procesów może więc zapisywać do deskryptora pliku p[1] i odczytywać z deskryptora pliku p[0]. Pojawia się tu jednak pewien problem. Potoki są przeznaczone do użytku jako jednokierunkowe kanaly komunikacyjne. Jeśli oba procesy swobodnie odczytują i zapisują do potoku w tym samym czasie, może powstać zamieszanie.



Rysunek 7.2 Drugi przykład potoku

Aby tego uniknąć, zwyczajowo każdy proces tylko odczytuje z potoku lub tylko do niego zapisuje i zamknie nie używany przez siebie deskryptor pliku. Program rzeczywiście musi to robić, aby uniknąć problemów, kiedy proces wysyłający zamknie swoją końcówkę zapisu (podrozdział 7.1.4 wyjaśnia dlaczego). Dotychczasowe przykłady działają tylko dla tego, że proces odbierający wie dokładnie, ilu danych oczekuje. Następny przykład pokazuje ukończone rozwiązanie:

```
/* trzeci przykład potoku */
#include <unistd.h>
#include <stdio.h>

#define MSGSIZE 16

char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
```

```

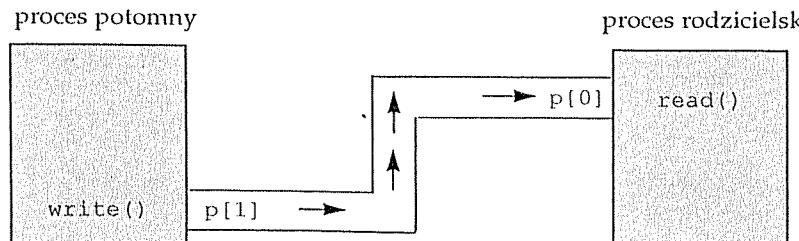
char *msg3 = "hello, world #3";
main()
{
    char inbuf[MSGSIZE];
    int p[2], j;
    pid_t pid;

    /* otwórz potok */
    if(pipe(p) == -1)
    {
        perror("pipe call");
        exit(1);
    }

    switch(pid = fork()){
    case -1:
        perror("fork call");
        exit(2);
    case 0:
        /* jeśli potomek, to zamknij deskryptor
         * odczytu pliku i zapisz do potoku */
        close(p[0]);
        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);
        break;
    default:
        /* jeśli rodzic, to zamknij deskryptor
         * zapisu pliku i odczytaj z potoku */
        close(p[1]);
        for(j = 0; j < 3; j++)
        {
            read(p[0], inbuf, MSGSIZE);
            printf("%s\n", inbuf);
        }
        wait(NULL);
    }
    exit(0);
}

```

Końcowym wynikiem jest jednokierunkowa droga między procesem rodzicielskim a procesem potomnym. Tę uproszczoną sytuację obrazuje rysunek 7.3.



Rysunek 7.3 Trzeci przykład potoku

Ćwiczenie 7.1 W ostatnim przykładzie potok był użyty do ustanowienia łącza między procesem rodzicielskim a procesem potomnym. W rzeczywistości deskryptory pliku dla potoku mogą być przekazywane przez kilka wywołań funkcji `fork`. Oznacza to, że więcej niż jeden proces może odczytywać z potoku i więcej niż jeden proces może zapisywać do potoku. Aby to zademonstrować, napisz program, który tworzy trzy procesy, z których dwa zapisują do potoku, a jeden odczytuje z potoku. Niech proces odczytu drukuje wszystkie komunikaty, które odbierze na swoim standardowym wyjściu.

Ćwiczenie 7.2 Aby ustanowić dwukierunkową komunikację między procesami, można utworzyć dwa potoki działające w różnych kierunkach. Obmyśl możliwą konwersację między dwoma procesami i zaimplementuj używając w ten sposób potoków.

7.1.3 Wielkość potoku

Dotychczas nasze przykłady transmitowały tylko małe ilości danych. W praktycznych zastosowaniach trzeba pamiętać o tym, że wielkość potoku jest ograniczona. Inaczej mówiąc w potoku może znajdować się tylko pewna liczba bajtów, a późniejsze zapisy będą blokowane. Minimalna wielkość jest zdefiniowana przez POSIX jako 512 bajtów. Jednak w rzeczywistości większość systemów umożliwia znacznie większe wartości. W trakcie programowania trzeba znać maksymalną wielkość potoku, ponieważ wpływa ona na operacje zapisu i odczytu. Jeśli wykonujemy zapis do potoku, w którym jest dostateczna ilość miejsca, dane są wysyłane do potoku, a wywołanie funkcji `write` wraca niezwłocznie. Jednak jeśli wykonujemy zapis, który przepelnią potok, wykonywanie procesu zostaje zawieszone, dopóki inny proces nie zrobi miejsca, odczytując dane z potoku.

W następnym przykładzie program zapisuje do potoku znak po znaku, dopóki wywołanie funkcji `write` nie zostanie zablokowane. Program wykorzystuje funkcję `alarm`, aby uchronić proces przed zbyt długim czekaniem po zablokowaniu zapisu na odczyt, który nigdy nie nastąpi. Zwróć uwagę na wykorzystanie procedury `fpathconf` (opisanej w podrozdziale 4.6.4), używanej do określenia maksymalnej liczby bajtów, które mogą być zapisane do potoku na jednym końcu.

```

/* zapisuj do potoku, dopóki nie zostanie zablokowany */

#include <signal.h>
#include <unistd.h>
#include <limits.h>

int count;
void alarm_action(int);

main()
{
    int p[2];
    int pipe_size;
    char c = 'x';

    if(fpathconf(p[1], _PC_PIPE_BUF) != -1)
        pipe_size = _PC_PIPE_BUF;
    else
        pipe_size = 512;
    if(pipe(p) == -1)
        perror("pipe error");
    if(alarm(1) == -1)
        perror("alarm error");
    if(write(p[1], &c, 1) == -1)
        perror("write error");
    if(count == pipe_size)
        alarm(0);
    if(read(p[0], &c, 1) == -1)
        perror("read error");
    if(c != 'x')
        alarm(0);
    if(count == pipe_size)
        alarm(0);
}

```

```

static struct sigaction act;

/* przygotuj program obsługi sygnału */
act.sa_handler = alarm_action;
sigfillset(&(act.sa_mask));

/* utwórz potok */
if(pipe(p) == -1)
{
    perror("pipe call");
    exit(1);
}

/* określ wielkość potoku */
pipe_size = fpathconf(p[0], _PC_PIPE_BUF);
printf("Maximum size of write to pipe: %d bytes\n", pipe_size);

/* ustaw program obsługi sygnału */
sigaction(SIGALRM, &act, NULL);

while(1)
{
    /* ustaw alarm */
    alarm(20);

    /* zapisz do potoku */
    write(p[1], &c, 1);

    /* wyłącz alarm */
    alarm(0);

    if((++count % 1024) == 0)
        printf("%d characters in pipe\n", count);
}

/* wywoływany po odebraniu SIGALRM */
void alarm_action(int signo)
{
    printf("write blocked after %d characters\n", count);
    exit(0);
}

```

W wielu systemach program ten wyświetli następujące wyniki:

Maximum size of write to pipe: 32768 bytes

1024 characters in pipe

2048 characters in pipe

3072 characters in pipe

4096 characters in pipe

5120 characters in pipe

31744 characters in pipe
32768 characters in pipe
write blocked after 32768 characters

Zwrót uwagę, ile razy większa jest rzeczywista granica w porównaniu z minimalną wielkością zalecaną przez POSIX.

Sprawa jest nieco bardziej skomplikowana, kiedy proces próbuje za pomocą pojedynczego write zapisać więcej danych, niż potok może otrzymać, nawet jeśli jest pusty. W tym przypadku jądro najpierw zapisze tyle danych do potoku, ile będzie mogło; później zawiesi wykonanie procesu, aż dostępne stanie się miejsce dla reszty danych. To jest ważne; zwykle zapis do potoku wykonuje się **niepodzielnie** (ang. *atomically*): wszystkie dane są przekazywane za pomocą pojedynczej, nieprzerwanej operacji jądra. Jeśli zapisujemy więcej danych, niż potok może zawierać, zapis musi być wykonywany etapami. Jeśli kilka współbieżnych procesów zapisuje w ten sposób do tego samego potoku, dane mogą zostać pomieszane.

Interakcja odczytu i potoku jest znacznie prostsza. Kiedy wywołujemy funkcję read, system sprawdza, czy potok jest pusty. Jeśli tak, read będzie (zwykle) blokowane, aż dane zostaną zapisane do potoku przez inny proces. Jeśli dane czekają w potoku, wtedy read wraca, nawet jeśli jest tam mniej danych niż żądana liczba.

7.1.4 Zamknięcie potoków

Co się dzieje, jeśli deskryptor pliku, który reprezentuje jeden koniec potoku, zostanie zamknięty? Mogą wystąpić dwa przypadki:

- Zamknięcie deskryptora pliku tylko do zapisu.** Jeśli istnieją inne procesy, które ciągle mają potok otwarty do zapisu, nic się nie dzieje. Jednak jeśli nie ma już więcej procesów zdolnych do zapisu do potoku, a potok jest pusty, każdy proces próbujący odczytać dane z potoku wraca bez danych. Procesy, które w uśpieniu czekają na odczyt z potoku, zostaną obudzone, a ich funkcje read zwrócią zero. Dlatego skutek dla procesów czytających przypomina osiągnięcie końca zwykłego pliku.
- Zamknięcie deskryptora pliku tylko do odczytu.** Jeśli istnieją jeszcze procesy, mające potok otwarty do odczytu, wtedy również nic się nie zdarzy. Jeśli jednak żaden inny proces nie czyta z potoku, do wszystkich procesów czekających na zapis do potoku będzie przez jądro wysłany sygnał SIGPIPE. Jeśli ten sygnał nie zostanie przechwycony, proces zakończy się. Gdy sygnał zostanie przechwycony, po zakończeniu procedury przerwania funkcja write zwróci -1, a zmenna errno powinna zawierać EPIPE. Do procesu, który spróbuje później zapisać do potoku, też będzie wysłany sygnał SIGPIPE.

7.1.5 Nie blokujące odczyty i zapisy

Jak widzieliśmy, przy używaniu potoku zarówno odczyt, jak i zapis, mogą być zablokowane. Jednak czasami nie jest to pożądane. Na przykład możesz chcieć, żeby program wykonał procedurę obsługi błędu lub przeglądał kilka potoków, aż otrzymasz dane z jednego z nich. Na szczęście istnieją dwa proste sposoby upewnienia się, że odczyt lub zapis do potoku nie ulegną zawieszeniu.

Pierwszą metodą jest użycie do potoku funkcji fstat. Pole st_size w zwracanej strukturze stat podaje liczbę znaków, znajdujących się aktualnie w potoku. Jeśli tylko jeden proces odczytuje z potoku, to wszystko w porządku. Jeśli jednak kilka procesów odczytuje z potoku, inne procesy mogą odczytać dane w przerwie między funkcjami fstat i read.

Drugą metodą jest (ponowne) użycie funkcji fcntl. Pełni ona wiele różnych ról, m.in. pozwala procesowi ustawić znacznik O_NONBLOCK dla deskryptora pliku. Powstrzymuje to następne wywołania funkcji read lub write odnoszące się do potoku przed blokowaniem. W tym kontekście funkcja fcntl może być użyta następująco:

```
#include <fcntl.h>
.

.

if(fcntl(filedes, F_SETFL, O_NONBLOCK) == -1)
    perror("fcntl");
```

Jeśli filedes jest deskryptorem pliku tylko do zapisu dla potoku, wtedy następne wywołania funkcji write nie będą nigdy blokowane, nawet jeśli potok jest pełny. W zamian zwracają natychmiast wartość -1 i ustawiają errno na EAGAIN. Podobnie, gdy filedes reprezentuje koniec potoku do odczytu, wtedy, jeśli nie ma żadnych danych w potoku, proces natychmiast zwraca wartość -1, zamiast ulec zawieszeniu. Podobnie jak przy wywołaniu write, errno będzie ustawione na EAGAIN. (Gdy zamiast tego znacznika zostanie ustawiony znacznik O_NDELAY, zachowanie read się zmieni. Zwróci 0 przy pustym potoku. Nie będziemy rozważać dalej tego przypadku).

Następujący program demonstruje te odmiany tematu fcntl. Tworzy on potok, umieszcza znacznik O_NONBLOCK dla deskryptora pliku do odczytu, a następnie wywołuje fork. Proces potomny wysyła komunikat do rodzica, czekającego w pętli odpytującej potok, aby zobaczyć, czy przybyły jakieś dane.

```
/* przykład O_NONBLOCK */

#include <fcntl.h>
#include <errno.h>
.

#define MSGSIZE 6

int parent(int *);
int child(int *);

char *msg1 = "hello";
```

```
char *msg2 = "bye!!";
main()
{
    int pfd[2];
    /* otwórz potok */
    if(pipe(pfd) == -1)
        fatal("pipe call");
    /* ustaw znacznik O_NONBLOCK dla p[0] */
    if(fcntl(pfd[0], F_SETFL, O_NONBLOCK) == -1)
        fatal("fcntl call");
    switch(fork()){
        case -1:
            /* błąd */
            fatal("fork call");
        case 0:
            /* potomek */
            child(pfd);
            default:
                /* rodzic */
                parent(pfd);
    }
}

int parent(int p[2])      /* kod dla rodzica */
{
    int nread;
    char buf[MSGSIZE];
    close(p[1]);
    for(;;)
    {
        switch(nread = read(p[0], buf, MSGSIZE)){
            case -1:
                /* sprawdź, czy nic nie ma w potoku */
                if(errno == EAGAIN)
                    {
                        printf("(pipe empty)\n");
                        sleep(1);
                        break;
                    }
                else
                    fatal("read call");
            case 0:
                /* potok został zamknięty */
                printf("End of conversation\n");
                exit(0);
            default:
                printf("MSG=%s\n",buf);
        }
    }
}

int child(int p[2])
{
    int count;
```

```

close(p[0]);

for(count = 0; count < 3; count++)
{
    write(p[1], msg1, MSGSIZE);
    sleep(3);
}

/* wyslij komunikat koncowy */
write(p[1], msg2, MSGSIZE);
exit(0);
}

```

Przykładowy program używa procedury obsługi błędu o nazwie `fatal`. Wprowadziliśmy ją w poprzednim rozdziale. Jest ona zaimplementowana następująco (podajemy ją tu ponownie, abyś nie musiał wertować książek):

```

int fatal(char *s)      /* drukuj komunikat bledu i zakończ */
{
    perror(s);
    exit(1);
}

```

Wyjście przykładowego programu nie jest całkowicie przewidywalne, ponieważ liczba komunikatów pipe empty może się zmieniać. Jednak na jednym z komputerów dał on następujący wynik:

```

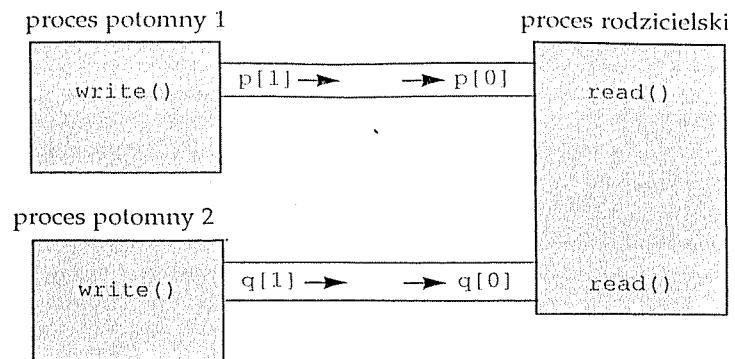
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
MSG=hello
(pipe empty)
(pipe empty)
(pipe empty)
MSG=bye!!
End of conversation

```

7.1.6 Wykorzystanie funkcji select do obsługi wielu potoków

Użycie nie blokujących odczytów i zapisów jest doskonale dla prostych aplikacji. Inne rozwiązanie, odpowiednie do manipulacji wieloma potokami równocześnie, polega na użyciu funkcji systemowej o nazwie `select`.

Wyobraź sobie, że proces rodzicielski działa jako proces serwera i może mieć dowolną liczbę komunikujących się z nim procesów klienta (potomnych), jak pokazano na rysunku 7.4.



Rysunek 7.4 Procesy klienta/serwera używające potoków

Proces serwera musi tu poradzić sobie z sytuacją, w której informacja nadchodzi więcej niż jednym potokiem. W dodatku, jeśli nic nie nadchodzi żadnym potokiem, proces serwera powinien zablokować się, aż coś nadjejdzie – bez ciągłego odpłytywania. Gdy informacja nadchodzi więcej niż jednym potokiem, proces serwera musi wiedzieć, którymi potokami, aby mógł uporać się z nimi w prawidłowej kolejności.

Funkcja systemowa, umożliwiająca to wszystko, ma nazwę `select` (istnieje również podobna funkcja o nazwie `poll`). Funkcja systemowa `select` jest używana nie tylko dla potoków, lecz także dla zwykłych plików, urządzeń terminala, plików FIFO (omawianych w podrozdziale 7.2) i gniazd (omawianych w rozdziale 10). Funkcja `select` działa za pomocą wskazywania, które z określonego zestawu deskryptorów pliku są gotowe do odczytu, do zapisu, lub wystąpiły w nich sytuacje błędne. Oczywiście, nie chcemy procesu serwera blokować na zawsze, jeśli żaden z tych warunków nie jest nigdy prawdziwy i dlatego `select` może także ustawić określony czas oczekiwania.

Użycie

```

#include <sys/time.h>
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *errorfds, struct timeval *timeout);

```

Pierwszy parametr, `nfds`, przekazuje do funkcji `select` liczbę deskryptorów pliku, którymi zainteresowany jest proces serwera. Na przykład jeśli deskryptory pliku 0, 1 i 2 są przypisane odpowiednio do `stdin`, `stdout` i `stderr`, a mamy otwarte dwa dalsze pliki, z przydzielonymi deskryptorami pliku 3 i 4, wtedy musimy ustawić `nfds` na 5. Jako programista możesz pracować z tymi wartościami albo użyć stałej `FD_SETSIZE` zdefiniowanej w `<sys/time.h>`. Stala `FD_SETSIZE` określa maksymalną liczbę deskryptorów pliku używanych w funkcji `select`.

Parametry 2, 3 i 4 są wskaźnikami do **masek bitowych** (ang. *bit masks*), w których każdy bit reprezentuje deskryptor pliku. Jeśli bit jest włączony, wskazuje zainteresowanie odnośnym deskryptorem pliku. Maska readfds pyta, czy jest coś do odczytu, writefds – czy któryś z deskryptorów pliku jest gotów przyjąć zapis, a errorfds – czy na którymś z podanych deskryptorów pliku nie wystąpił warunek wyjątkowy, na przykład czy połączeniem sieciowym nadeszły dane poza pasmem. Ponieważ manipulacja bitami jest raczej brzydka i potencjalnie nieprzewidziana, dostarczany jest abstrakcyjny typ danych `fd_set`, razem z makroinstrukcjami (lub funkcjami, w zależności od implementacji) do manipulacji tego typu wystąpieniami. Makroinstrukcje manipulacji bitami pokazano poniżej:

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

/* inicjuj maskę wskazywaną przez fdset */
void FD_ZERO(fd_set *fdset);

/* ustaw bit fd w masce wskazywanej przez fdset */
void FD_SET(int fd, fd_set *fdset);

/* czy bit fd jest ustawiony w masce wskazywanej przez fdset */
int FD_ISSET(int fd, fd_set *fdset);

/* wylacz bit fd w masce wskazywanej przez fdset */
void FD_CLR(int fd, fd_set *fdset);
```

Następujący przykład pokazuje, jak wyrazić zainteresowanie dwoma deskryptorami otwartych plików:

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

.

.

int fd1, fd2;
fd_set readset;

fd1 = open("file1", O_RDONLY);
fd2 = open("file2", O_RDONLY);

FD_ZERO(&readset);
FD_SET(fd1, &readset);
FD_SET(fd2, &readset);

switch(select(5, &readset, NULL, NULL, NULL))
{
    /* logika programu */
}
```

Aby zrozumieć ten przykład, należy pamiętać, że `fd1` i `fd2` to małe liczby całkowite, reprezentujące pierwszą wolną pozycję deskryptorów pliku i używane jako indeksy masek bitowych. Zauważ, że argumenty `writefds` i `errorfds` funkcji

`select` są ustawione na `NULL`. Oznacza to, że jesteśmy zainteresowani tylko odczytem z `fd1` i `fd2`.

Piąty parametr funkcji `select`, `timeout`, jest wskaźnikiem do struktury `timeval`, będącej następującą strukturą:

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

struct timeval {
    long tv_sec;           /* sekundy */
    long tv_usec;          /* i mikrosekundy */
};
```

Jeśli wskaźnik jest pusty, jak w naszym przykładzie, funkcja `select` blokuje się zawsze lub dopóki nie pojawi się coś interesującego. Jeśli `timeout` wskazywany w strukturze zawiera 0 sekund, wtedy `select` wraca natychmiast (bez blokowania). Na koniec, jeśli struktura `timeout` zawiera wartość niezerową, `select` powróci po określonej liczbie sekund lub mikrosekund, o ile nie ma żadnej aktywności deskryptorów pliku.

Wartością zwracaną przez `select` jest -1 w przypadku błędu, 0 po czasie oczekiwania albo liczba całkowita wskazująca numer interesującego nas deskryptora pliku. Jednak należy się tu słowo ostrzeżenia: gdy `select` wraca, ponownie ustawia maski bitowe wskazywane odpowiednio przez `readfds`, `writefds` albo `errorfds`, przez zerowanie masek i ustawienie deskryptorów pliku, które zawierają poszukiwaną informację. Dlatego istotne jest trzymanie kopii swojej oryginalnej maski.

Oto bardziej złożony przykład, oparty na użyciu trzech potoków do połączenia trzech procesów potomnych. Proces rodzicielski zerka także na standardowe wejście.

```
/* server -- tworzy trzy procesy potomne i obsługuje je */
#include <sys/time.h>
#include <sys/wait.h>

#define MSGSIZE 6

char *msg1 = "hello";
char *msg2 = "bye!!";

void parent(int ***);
int child(int *);

main()
{
    int pip[3][2];
    int i;

    /* utwórz trzy potoki komunikacyjne i trzy potomki */
    for(i = 0; i < 3; i++)
    {
        if(pipe(pip[i]) == -1)
            fatal("pipe call");
    }
```

```

switch(fork()){
    case -1:           /* błąd */
        fatal("fork call");
    case 0:            /* potomek */
        child(pip[i]);
    }

parent(pip);

exit(0);
}

/* rodzic nasłuchuje na wszystkich trzech potokach */
void parent(int p[3][2])      /* kod rodzica */
{
    char buf[MSGSIZE], ch;
    fd_set set, master;
    int i;

    /* zamknij wszystkie niepotrzebne deskryptory pliku do zapisu */
    for(i = 0; i < 3; i++)
        close(p[i][1]);

    /* ustaw maski bitowe dla funkcji systemowej select */
    FD_ZERO(&master);
    FD_SET(0, &master);
    for(i = 0; i < 3; i++)
        FD_SET(p[i][0], &master);

    /* funkcja select jest wywoływana bez czasu oczekiwania,
     * powinna się blokować aż do wystąpienia zdarzenia */
    while(set = master, select(p[2][0]+1, &set, NULL, NULL, NULL) > 0)
    {
        /* nie możemy zapomnieć informacji ze standardowego wejścia,
         * tj. fd=0 */
        if(FD_ISSET(0, &set))
        {
            printf("From standard input...");
            read(0, &ch, 1);
            printf("%c\n", ch);
        }

        for(i = 0; i < 3; i++)
        {
            if(FD_ISSET(p[i][0], &set))
            {
                if(read(p[i][0], buf, MSGSIZE)>0)
                {
                    printf("Message from child%d\n", i);
                    printf("MSG=%s\n", buf);
                }
            }
        }
    }
}

```

```

    /* serwer powinien powrócić do programu głównego,
     * jeśli nie ma już dzieci */
    if(waitpid(-1, NULL, WNOHANG) == -1)
        return;
}

int child(int p[2])
{
    int count;

    close(p[0]);

    for(count = 0; count < 2; count++)
    {
        write(p[1], msg1, MSGSIZE);
        /* czekaj losową ilość czasu */
        sleep(getpid() % 4);
    }

    /* wyślij końcowy komunikat */
    write(p[1], msg2, MSGSIZE);
    exit(0);
}

```

A to typowe wyjście po uruchomieniu tego programu:

Message from child 0
 MSG=hello
 Message from child 1
 MSG=hello
 Message from child 2
 MSG=hello

d (użytkownik wprowadził [d], a później [Return])
 From standard input *d* (echo znaku [d])
 From standard input (echo znaku [Return])

Message from child 0
 MSG=hello
 Message from child 1
 MSG=hello
 Message from child 2
 MSG=hello

Message from child 0
 MSG=bye
 Message from child 1
 MSG=bye
 Message from child 2
 MSG=bye

Zwróć uwagę, że w tym przykładzie użytkownik wprowadził literę d, a następnie powrót karetki, co zostało dostrzeżone przez funkcję select na standardowym wejściu.

7.1.7 Potoki i funkcja systemowa exec

Pamiętasz zapewne, że pomiędzy dwoma programami na poziomie powłoki może być ustawiony potok:

```
$ ls | wc
```

Jak to jest robione? Odpowiedź składa się z dwóch części. Po pierwsze, powłoka wykorzystuje fakt, że otwarte deskryptory plików przechodzą otwarte (domyślnie) przez wywołania exec. To oznacza, że dwa deskryptory plików dla potoków otwarte przed kombinacją fork/exec pozostaną otwarte, kiedy proces potomny rozpocznie wykonywanie nowego programu. Po drugie, przed wywołaniem exec, powłoka łączy standardowe wyjście ls z końcem potoku przeznaczonym do zapisu i standardowe wejście wc z końcem do odczytu. Może to być zrobione za pomocą użycia fcntl albo dup2, jak pokazano w ćwiczeniu 5.10. Ponieważ standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach mają odpowiednio wartości 0, 1 i 2, programista może na przykład połączyć następujące standardowe wyjście do innego deskryptora pliku (zakładając, że standardowe wejście jest otwarte), używając dup2. Zauważ, że dup2 zamknuje plik reprezentowany przez jego drugi parametr przed powtórnym przydziałem.

```
/* fcntl powinien powieść deskryptor pliku "1" */
dup2(filedes, 1);
.

.

.

/* program powinien teraz zapisywać swoje standardowe wyjście */
/* do pliku, do którego odnosi się filedes */
.

.

.
```

Nasz następny przykład, join, pokazuje w uproszczonej formie używany przez powłokę mechanizm potokowy. Program join pobiera dwa parametry, com1 i com2, z których każdy opisuje polecenie, mające być uruchomione. Oba parametry w rzeczywistości są tablicami wskaźników znakowych, które będą przekazane do execvp.

Program join powinien uruchomić oba programy i połączyć potokiem standardowe wyjście com1 ze standardowym wejściem com2. Działanie join jest pokazane na rysunku 7.5 i może być opisane w formie pseudokodu (wyłączając obsługę błędów) w sposób następujący:

*proces rozwidla się, rodzic czeka na proces potomny
i potomek działa*

potomek tworzy potok

następnie potomek rozwidla się

w potomku utworzonym przez drugie rozwidlenie (wnuczku):

standardowe wyjście jest łączone
za pomocą dup2 z końcem do zapisu potoku

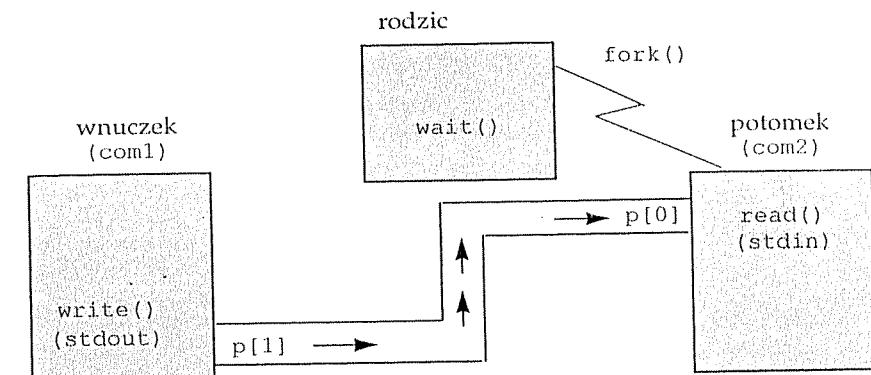
zamykane są nadmiarowe deskryptory plików

wykonywany jest program opisany przez com1

w potomku pierwszego rozwidlenia:
standardowe wejście jest łączone
za pomocą dup2 z końcem do odczytu potoku

zamykane są nadmiarowe deskryptory plików

wykonywany jest program opisany przez com2



Rysunek 7.5 Program join

Rzeczywista implementacja join jest następująca (ponownie używamy fatal, jak w podrozdziale 7.1.5):

```
/* join -- łączy dwa polecenia za pomocą potoku */

int join(char *com1[], char *com2[])
{
    int p[2], status;

    /* utwórz proces potomny do uruchomienia polecenia */
    switch(fork()){
        case -1:           /* błąd */
            fatal("1st fork call in join");
        case 0:             /* potomek */
            break;
        default:           /* rodzic */
            wait(&status);
            return (status);
    }
}
```

```

/* pozostałe procedury wykonywane przez proces potomny */
/* utwórz potok */
if(pipe(p) == -1)
    fatal("pipe call in join");

/* utwórz inny proces */
switch(fork()){
case -1:
    /* błąd */
    fatal("2nd fork call in join");
case 0:
    /* proces zapisu */
    dup2(p[1], 1);      /* podłącz standardowe wyjście do potoku */

    close(p[0]);        /* zamknij deskryptory pliku */
    close(p[1]);

    execvp(com1[0], com1);
    /* jeśli execvp powróciło, wystąpił błąd */
    fatal("1st execvp call in join");
default:
    /* proces odczytu */
    dup2(p[0], 0);      /* podłącz standardowe wejście do potoku */

    close(p[0]);
    close(p[1]);
    execvp(com2[0], com2);
    fatal("2nd execvp call in join");
}

```

Ta procedura może być wywoływana za pomocą następujących linii:

```

#include <stdio.h>

main()
{
    char *one[4] = {"ls", "-l", "/usr/lib", NULL};
    char *two[3] = {"grep", "^d", NULL};
    int ret;

    ret = join(one, two);
    printf("join returned %d\n", ret);
    exit(0);
}

```

Ćwiczenie 7.3 Jak uogólnić technikę pokazaną w join, aby połączyć w linię potoku więcej niż dwa polecenia?

Ćwiczenie 7.4 Włącz potoki do procesora poleceń smallsh, wprowadzonego w poprzednim rozdziale.

Ćwiczenie 7.5 Wymyśl metodę, w której proces tworzy proces potomny w celu uruchomienia pojedynczego programu, rodzinę czyta standardowe wyjście potom-

ka za pomocą potoku. Warto zauważyć, że ta idea leży u podstaw procedur popen i pclose, które stanowią część Standardowej Biblioteki I/O. Uwalniają one programistę od troski o większość szczegółów koordynacji funkcji fork, exec, close, open lub dup2. Będziemy omawiać te procedury w rozdziale 11.

7.2 Pliki FIFO lub nazwane potoki

Potoki są eleganckim i połącznym mechanizmem komunikacji międzyprocesowej. Jednak mają one kilka wad.

Po pierwsze i najważniejsze, potok może być używany tylko do łączenia procesów, mających wspólnych przodków, jak proces rodzicielski i jego proces potomny. Ta wada staje się widoczna, kiedy próbujemy opracować prawdziwy program serwera, który egzystuje na stale, dostarczając usług systemowych. Przykłady zawierają serwer sterujący siecią i buforami wydruku. W idealnym przypadku proces klienta powinien zaistnieć, skomunikować się z nie związanym procesem serwera za pomocą potoku, a następnie oddalić się ponownie. Niestety, nie można tego zrobić, używając konwencjonalnych potoków.

Po drugie, potoki nie mogą być trwale. Muszą być tworzone za każdym razem, gdy są potrzebne, i usuwane, gdy kończy się korzystający z nich proces.

Niedostatki te niweluje pewna dostępna odmiana potoku. Ten mechanizm komunikacji międzyprocesowej jest nazywany **plikiem FIFO** albo **nazwanym potokiem** (ang. *named pipe*). Dopuski dotyczy to odczytu i zapisu, pliki FIFO są identyczne z potokami, działając jako kanał komunikacyjny typu pierwszy na wejściu-pierwszy na wyjściu między procesami. Istotnie, pliki FIFO i potoki najczęściej mają wiele wspólnego kodu na poziomie jądra. Jednak w przeciwieństwie do potoku, FIFO jest trwałym elementem i posiada nazwę pliku Uniksa. Plik FIFO ma też właściciela, wielkość i powiązane prawa dostępu. Może być otwierany, zamknięty i usuwany jak każdy inny plik Uniksa, ale przy odczytzie lub zapisie wyświetla własności identyczne do potoków.

Przed badaniem interfejsu programowego spójrzmy na użycie FIFO na poziomie polecień. Do tworzenia pliku FIFO używane jest polecenie mknod (zgadnij, co oznacza p):

```
$ /etc/mknod channel p
```

Argument channel jest tu nazwą FIFO (może być zastąpiony jakkolwiek ważną nazwą ścieżki Uniksa). Drugi argument, p, mówi, że polecenie mknod utworzyło plik FIFO. Argument ten jest potrzebny, ponieważ mknod używa się też do tworzenia plików urządzeń.

Teraz utworzony plik FIFO może być identyfikowany za pomocą polecenia ls w następujący sposób:

```
$ ls -l channel
prw-rw-r--- 1 ben usr 0 Aug 1 21:05 channel
```

Litera p w pierwszej kolumnie wydruku wskazuje, że channel jest plikiem typu FIFO. Zauważ, jakie channel ma prawa dostępu (odczyt/zapis dla właściciela i grupy właściciela, tylko odczyt dla wszystkich innych); właściciela i grupę właściciela (ben, usr); wielkość (0 bajtów, więc jest aktualnie pusty) i datę utworzenia. Plik FIFO może być odczytywany i zapisywany za pomocą standardowych poleceń Uniksa, na przykład:

```
$ cat < channel
```

Jeśli to polecenie jest wykonywane zaraz po utworzeniu channel, może się zawiążeć. Dzieje się tak, ponieważ domyślnie proces otwierania pliku FIFO dla odczytu będzie blokowany, dopóki inny proces nie spróbuje otworzyć FIFO do zapisu. Podobnie, proces próbujący otworzyć plik FIFO do zapisu będzie blokowany, dopóki jakiś proces nie spróbuje otworzyć FIFO do odczytu. Jest to całkiem rozsądne, ponieważ oszczędza zasoby systemu i upraszcza koordynację programu. W konsekwencji, jeśli chcemy utworzyć proces czytający i proces zapisujący dla naszego ostatniego przykładu, musimy uruchomić jeden z procesów w tle (lub co najmniej w innym oknie interfejsu GUI), na przykład:

```
$ cat < channel &
```

```
102
```

```
$ ls -l > channel; wait
```

```
total 17
```

	1	ben	usr	0	Aug	1	21:05	channel
-rw-rw-r--	1	ben	usr	0	Aug	1	21:06	f
-rw-rw-r--	1	ben	usr	937	Jul	27	22:30	fifos
-rw-rw-r--	1	ben	usr	7152	Jul	27	22:11	pipes.cont

Przeanalizujmy to. Wydruk katalogu jest początkowo tworzony przez ls i wtedy zapisywany do FIFO. Następnie czekające polecenie cat odczytuje dane z pliku FIFO i wyświetla na ekranie. Proces uruchamiający cat teraz się kończy. Dzieje się tak, bo gdy plik FIFO nie jest już dłużej otwarty do zapisu, odczyt z niego będzie zwracać 0 jak w normalnym potoku, co cat przyjmie za koniec pliku. Natomiast polecenie wait powoduje, że powłoka czeka na zakończenie cat przed ponownym wyświetleniem znaku zatrzymania.

7.2.1 Programowanie za pomocą plików FIFO

Przeważnie programowanie z użyciem plików FIFO jest identyczne z programowaniem za pomocą zwykłych potoków. Jedyna znacząca różnica to inicjacja. Zamiast użycia pipe, plik FIFO jest tworzony za pomocą mkfifo. W starszych wersjach Uniksa być może będziesz musiał użyć bardziej ogólnego wywołania o nazwie mknod.

Użycie

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Funkcja systemowa mkfifo tworzy plik FIFO o nazwie podanej przez pierwszy parametr pathname. Plik FIFO będzie miał dane uprawnienie mode. To uprawnienie będzie modyfikowane przez wartość umask procesu.

Po utworzeniu, plik FIFO musi być otwarty za pomocą open. Przykładowy fragment:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
.
.
mkfifo("/tmp/fifo", 0666);
.
.
fd = open("/tmp/fifo", O_WRONLY);
```

otwiera więc plik FIFO do zapisu. Funkcja open będzie blokowana, dopóki inny proces nie otworzy FIFO do odczytu (oczywiście, jeśli plik FIFO już był otwarty do odczytu, nasze wywołanie open wróci natychmiast).

Możliwe jest także nie blokujące się wywołanie funkcji open dla pliku FIFO. Aby tego dokonać, wywołanie funkcji open musi być robione ze znacznikiem O_NONBLOCK (zdefiniowanym w <fcntl.h>) połączonym z O_RDONLY albo O_WRONLY, na przykład:

```
if((fd = open("/tmp/fifo", O_WRONLY | O_NONBLOCK)) == -1)
    perror("open on fifo");
```

Jeśli żaden proces nie ma otwartego pliku FIFO dla odczytu, open będzie zwracać -1 zamiast blokowania się, a errno będzie zawierać ENXIO. Jeśli z kolei wywołanie funkcji open było pomyślne, następne wywołania write w odniesieniu do FIFO też nie będą się blokować.

Nadszedł już czas na przykład. Przedstawiamy dwa programy, które pokazują, jak można użyć pliku FIFO do implementacji systemu komunikatów. Wykorzystują one fakt, że wywołania funkcji read i write w odniesieniu do pliku FIFO są niepodzielne. Jeśli komunikaty o stałej wielkości są przekazywane za pomocą pliku FIFO, indywidualne komunikaty pozostaną nietknięte przy odczycie, nawet jeżeli kilka procesów jednocześnie zapisuje do potoku.

Najpierw przyjrzymy się programowi sendmessage, który wysyła indywidualne komunikaty do pliku FIFO o nazwie fifo. Wywołuje się go następująco:

```
$ sendmessage 'message text 1' 'message text 2'
```

Zwróć uwagę, że każdy komunikat jest zamknięty w cudzysłowach i liczony jako jeden długi argument. Jeśli tak nie zrobimy, każde słowo będzie traktowane jako oddzielnny komunikat. Kod programu sendmessage wygląda następująco:

```
/* sendmessage – wysyła komunikaty przez FIFO */
.
.
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#define MSGSIZ 63
```

```

char *fifo = "fifo";
main(int argc, char **argv)
{
    int fd, j, nwrite;
    char msgbuf[MSGSZ+1];

    if(argc < 2)
    {
        sprintf(stderr, "Usage: sendmessage msg ... \n");
        exit(1);
    }

    /* otwórz fifo z ustawionym O_NONBLOCK */
    if((fd = open(fifo, O_WRONLY | O_NONBLOCK)) < 0)
        fatal("fifo open failed");

    /* wyslij komunikat */
    for(j = 1; j < argc; j++)
    {
        if(strlen(argv[j]) > MSGSZ)
        {
            sprintf(stderr, "message too long %s\n", argv[j]);
            continue;
        }

        strcpy(msgbuf, argv[j]);

        if((nwrite = write(fd, msgbuf, MSGSZ+1)) == -1)
            fatal("message write failed");
    }
    exit(0);
}

```

Ponownie używamy naszej procedury błędu `fatal`. Komunikaty są wysyłane jako 64-znakowe kawałki, za pomocą nie blokowanych wywołań funkcji `write`. Rzeczywisty tekst komunikatu jest ograniczany do 63 znaków, aby uwzględnić końcowy znak zera.

Program, który odbiera komunikaty za pomocą odczytu pliku FIFO, jest nazwany `rcvmessage`. Nie robi on nic użytecznego, i w zamierzeniu służy jako podstawowy szkielet.

```

/* rcvmessage -- odbiera komunikaty przez fifo */

#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#define MSGSZ     63

char *fifo = "fifo";

main(int argc, char **argv)
{
    int fd;
    char msgbuf[MSGSZ+1];

```

```

/* utwórz fifo, jeśli jeszcze nie istnieje */
if(mkfifo(fifo, 0666) == -1)
{
    if(errno != EEXIST)
        fatal("receiver: mkfifo");

    /* otwórz fifo do odczytu i zapisu */
    if((fd = open(fifo, O_RDWR)) < 0)
        fatal("fifo open failed");

    /* odbierz komunikaty */
    for(;;)
    {
        if(read(fd, msgbuf, MSGSZ+1) < 0)
            fatal("message read failed");

        /*
         * drukuje komunikat; w rzeczywistości
         * może robić coś bardziej
         * interesującego
         */
        printf("message received:%s\n", msgbuf);
    }
}

```

Zwróć uwagę, że plik FIFO jest otwierany dla odczytu i zapisu (przez znacznik `O_RDWR`). Aby zrozumieć, dlaczego tak się dzieje, załóżmy, że plik FIFO był otwarty ze znacznikiem tylko `O_RDONLY`. Program powinien najpierw zablokować się przy wywoaniu `open`. Jak tylko program `sendmessage` otworzy FIFO do zapisu, wywołanie `open` powróci; wtedy `rcvmessage` odczyta każdy wysłany komunikat. Jednak, gdyby plik FIFO był pusty i proces `sendmessage` zanikł, funkcja `read` zaczęłaby zwracać 0 natychmiast po wywołaniu, ponieważ żaden proces nie miałby pliku FIFO otwartego do zapisu. Dlatego program wszedłby w niepotrzebną pętlę. Użycie `O_RDWR` zapewnia, że co najmniej jeden proces, to znaczy sam `rcvmessage`, ma plik FIFO otwarty do zapisu. W wyniku wywołanie `read` będzie zawsze blokowane, dopóki dane są faktycznie zapisywane do pliku FIFO.

Następujący dialog pokazuje, jak można używać tych programów. Program `rcvmessage` jest umieszczany w tle, aby przyjmować komunikaty od różnych wywołań `sendmessage`.

```

$ rcvmessage &
40
$ sendmessage 'message 1' 'message 2'
message received: message 1
message received: message 2
$ sendmessage 'message number 3'
message received: message number 3

```

Ćwiczenie 7.6 Programy sendmessage i recvmessage tworzą podstawę prostego systemu buforowania. Na przykład komunikat wysyłany do recvmessage może być nazwą pliku, który jest w pewien sposób przetwarzany. Problem stanowi to, że bieżące katalogi sendmessage i recvmessage mogą być różne i względne nazwy ścieżek będą źle interpretowane. Jak można temu zapobiec? Czy użycie samego pliku FIFO jest odpowiednie, powiedzmy, dla buforowania wydruku w wielkim systemie?

Ćwiczenie 7.7 Jeśli program recvmessage został zastąpiony przez prawdziwy program serwera, zwykle chcemy być pewni, że jednocześnie działa tylko jedna kopia serwera. Istnieje kilka sposobów, aby to osiągnąć. Jeden z nich polega na tworzeniu pliku blokującego. Rozważ następującą procedurę:

```
#include <errno.h>
#include <fcntl.h>

extern int errno;
char * lck = "/tmp/lockfile";

int makelock(void)
{
    int fd;
    if((fd = open(lck, O_RDWR | O_CREAT | O_EXCL, 0600)) < 0)
    {
        if(errno == EEXIST)
            exit(1); /* coś innego już działa */
        else
            exit(127); /* nieoczekiwany błąd */
    }
    /* jeśli jesteśmy tu, to lock utworzony, więc powrót */
    close(fd);
    return (0);
}
```

Procedura ta wykorzystuje fakt, że wywołanie funkcji open jest niepodzielne. Tak więc, jeśli kilka procesów ściga się, aby wykonywać makelock, ten, który będzie pierwszy, utworzy plik blokujący i zablokuje pozostałe. Dodaj tę procedurę do sendmessage. Upewnij się, że gdy usuwasz program sendmessage za pomocą sygnałów SIGHUP albo SIGTERM, przed zakończeniem kasuje on plik blokujący. Jak sądzisz, dlaczego w procedurze makelock używamy open zamiast creat?

ROZDZIAŁ 8

Zaawansowana komunikacja międzyprocesowa

- 8.1 Wprowadzenie
- 8.2 Blokowanie rekordu
- 8.3 Zaawansowane urządzenia IPC

8.1 Wprowadzenie

Używając narzędzi opisanych w rozdziałach 6 i 7 możesz stosować podstawową komunikację międzyprocesową. Omawiane w tym rozdziale zaawansowane urządzenia komunikacji międzyprocesowej pozwalają na korzystanie z bardziej wyrafinowanych technik programowania.

Pierwszą i najprostszą z nich stanowi **blokowanie rekordu** (ang. *record locking*). W rzeczywistości nie jest to forma bezpośredniej komunikacji procesów, lecz raczej metoda ich współpracy. (Dlatego też początkowo chcieliśmy umieścić omówienie blokowania rekordu w rozdziale poświęconym strukturze plików. Jednak po namyśle stwierdziliśmy, że w tym miejscu pasuje lepiej). Pozwala ona procesowi czasowo rezerwować część pliku wyłącznie dla własnego użytku, rozwiązując w ten sposób kilka trudnych problemów zarządzania bazami danych. Z jednym zastrzeżeniem: XSI obsługuje tylko *doradce* blokowanie, co oznacza, że do procesu należy sprawdzenie, czy blokada została ustawiona w pliku.

Inne mechanizmy komunikacji międzyprocesowej, omawiane w tym rozdziale, są raczej egzotyczne. Te zaawansowane własności ogólnie opisuje się jako **urządzenia IPC** (ang. *IPC facilities*), gdzie IPC oznacza *komunikację międzyprocesową* (ang. *inter-process communication*). Ten pojedynczy termin opisowy uwydatnia podobieństwo struktur i użycia, chociaż zebrane są tu trzy różne typy urządzeń:

1. **Przekazywanie komunikatów** (ang. *message passing*). Urządzenie przekazywania komunikatów pozwala procesowi wysyłać i odbierać komunikaty, będące zasadniczo dowolną sekwencją bajtów lub znaków.

2. *Semafora* (ang. *semaphores*). W porównaniu z przekazywaniem komunikatów semafory dostarczają raczej niskopoziomowego sposobu synchronizacji procesów, niezbyt odpowiedniego dla przesyłania wielkich ilości informacji. Mają swoje teoretyczne podstawy w pracach E.W. Dijkstry (1968).
3. *Pamięć wspólna* (ang. *shared memory*). To najszybsze urządzenie IPC pozwala dwóm albo większej liczbie procesów wspólnie korzystać z danych zawartych w określonych segmentach pamięci. Oczywiście zwykle obszar danych procesu jest dla niego prywatny.

8.2 Blokowanie rekordu

8.2.1 Motywacja

Na początku zajmiemy się prostym przykładem demonstrującym wagę blokowania rekordu w pewnych sytuacjach.

Nasz przykład dotyczy dobrze znanej korporacji, *ACME Airlines*, używającej systemu uniksowego dla swojego systemu rezerwacji. Ma ona dwa biura rezerwacji, o nazwach A i B, z których każde ma własny terminal połączony z komputerem linii lotniczej. Pracownicy zajmujący się rezerwacją używają programu o nazwie *acmebook* do dostępu do bazy danych rezerwacji, zaimplementowanej jako zwykły plik Uniksa. Ten program pozwala użytkownikowi odczytywać i aktualizować bazę danych. W szczególności, urzędnik rezerwujący może zmniejszyć o jeden liczbę wolnych miejsc dla konkretnego lotu, zaznaczając, że rezerwacja została zrobiona.

Teraz przypuśćmy, że na lot ACM501 do Londynu pozostało tylko jedno wolne miejsce i że pani Jones wchodzi do biura A w tym samym czasie, co pan Smith wchodzi do biura B. Oboje pytają o miejsce na lot ACM501. Wtedy możliwa jest następująca seria zdarzeń:

1. Urzędnik w biurze A uruchamia program *acmebook*. Nazwijmy powstały proces *PA*.
2. W chwilę później urzędnik w biurze B też uruchamia *acmebook*. Nazwijmy ten proces *PB*.
3. Proces *PA* odczytuje teraz stosowną część bazy danych, używając funkcji systemowej *read*. Odkrywa on, że jest jeszcze jedno miejsce wolne.
4. Proces *PB* odczytuje bazę danych zaraz po *PA*. On również odkrywa, że jest jedno miejsce wolne na lot ACM501.
5. Proces *PA* ustawia wtedy liczbę wolnych miejsc dla lotu na zero, używając funkcji *write* do zmiany stosownej części bazy danych. Urzędnik w biurze A daje bilet pani Jones.
6. Chwilę później, proces *PB* też dokonuje zapisu w bazie danych, ponownie wstawiając zero w liczniku wolnych miejsc. Jednak tym razem ta wartość jest błędna; właściwie powinna wynosić -1. Wydatnia się tu, że chociaż *PA* już zaktualizował bazę danych, *PB* nie ma żadnego sposobu poznania tego i wciska się, jakby miejsce było jeszcze dostępne. W konsekwencji pan Smith także otrzymuje bilet i na ten lot zostało sprzedanych za dużo biletów.

Problem powstał, ponieważ plik Uniksa może być udostępniony równocześnie dowolnej liczbie procesów. Operacja logiczna, która składa się z szeregu wywołań funkcji *lseek*, *read* i *write*, może być wykonana przez dwa albo więcej procesów jednocześnie i mieć, jak widzieliśmy w naszym prostym przykładzie, zgubne skutki.

Jedno z rozwiązań to pozwolenie procesowi na *blokowanie* części pliku, na której on pracuje. Blokada, która nie zmienia w żaden sposób zawartości pliku, jest dla innych procesów znakiem, że dane, o których mowa, są w użyciu. Zapobiega to zakłóceniu przez inne procesy szeregu odeswanych fizycznych operacji, składających się na jedno działanie logiczne albo transakcję. Ten rodzaj mechanizmu jest często nazywany *blokowaniem rekordu* (ang. *record locking*), gdzie rekord oznacza dowolną część pliku. Aby było to całkowicie bezpieczne, sama operacja blokowania musi być niepodzielna, gdyż nie zazębia się wtedy z kolidującą próbą blokady z innego procesu.

Aby blokowanie działało, musi być wykonywane w jakiś decentralizowany sposób. Być może najlepiej jest, gdy blokowaniem zajmuje się jądro, chociaż do tego celu może również służyć proces użytkownika, działający jako agent bazy danych. Blokowanie rekordu w oparciu o jądro może być wykonane za pomocą naszej dobrej znajomej funkcji *fcntl*.

Zwróci uwagę, że pierwsze wydanie tej książki zawierało też procedurę o nazwie *lockf*, stanowiącą alternatywny sposób blokowania rekordu.¹ Można ją ciągle jeszcze znaleźć w wielu systemach – sprawdź szczegóły w swoim lokalnym podręczniku.

8.2.2 Blokowanie rekordu za pomocą funkcji *fcntl*

Spotykaliśmy już funkcję systemową kontroli pliku *fcntl*. Oprócz zwykłego wykorzystania funkcja *fcntl* może być używana do wykonania ogólnej formy blokowania rekordu. Oferuje ona dwa rodzaje blokady:

1. *Blokady odczytu* (ang. *read locks*). Blokada odczytu po prostu przeszkadza innym procesom w zastosowaniu innego rodzaju blokady *fcntl*, nazywanej blokadą zapisu. Kilka procesów może równocześnie zastosować blokadę odczytu do tego samego segmentu pliku. Na przykład blokady odczytu mogą być użyteczne, jeśli programista chce zabezpieczyć dane przed aktualizacją, ale nie chce ich ukryć przed badaniem przez innych użytkowników.
2. *Blokady zapisu* (ang. *write locks*). Blokada zapisu powstrzymuje wszystkie inne procesy przed stosowaniem blokady odczytu lub zapisu do pliku. Inaczej mówiąc, dla danego segmentu pliku może istnieć jednocześnie tylko jedna blokada zapisu. Na przykład blokada zapisu może być używana do usunięcia segmentu pliku przed widokiem publicznym w czasie wykonywania aktualizacji.

Ponownie musimy podkreślić, że stosownie do XSI, blokady dostępne za pomocą *fcntl* mają tylko charakter doradczy. Dlatego proces musi jawnie współpracować, aby blokada *fcntl* była skuteczna.

1. Por. przyp. ze s. VII.

Przy stosowaniu do blokady rekordu, funkcja fcntl jest używana następująco:

Użycie

```
#include <fcntl.h>
int fcntl(int filedes, int cmd, struct flock *ldata);
```

Jak zwykle, argument filedes musi być ważnym, otwartym deskryptorem pliku. Przy blokadzie odczytu filedes musi być otwarty przy użyciu O_RDONLY albo O_RDWR, więc nie może to być deskryptor pliku z funkcji creat. Przy blokadzie zapisu filedes musi być otwarty przy użyciu O_WRONLY albo O_RDWR.

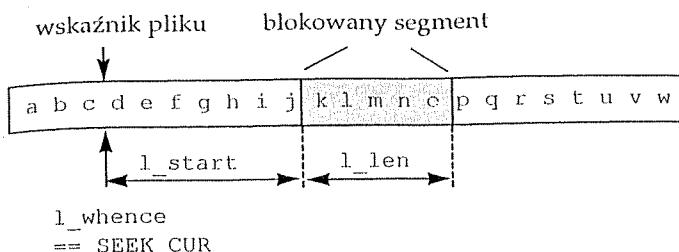
Jak widzieliśmy w naszych poprzednich spotkaniach z fcntl, parametr cmd określa działanie za pomocą wartości zdefiniowanej w <fcntl.h>. Z blokowaniem rekordu wiążą się następujące trzy wartości:

- | | |
|----------|--|
| F_GETLK | Pobiera opis blokady w oparciu o dane przekazywane przez argument ldata. (Zwracana informacja opisuje pierwszą blokadę, która „zabrania” blokady opisanej w ldata). |
| F_SETLK | Stosuje blokadę do pliku; wraca natychmiast, jeśli nie jest to możliwe. Używana także do usuwania aktywnej blokady. |
| F_SETLKW | Stosuje blokadę do pliku, wstrzymuje wykonanie (usypia), jeśli blokada została założona poprzednio przez inny proces. Proces wstrzymany (uśpiony) na blokadzie fcntl może być przerwany za pomocą sygnału. |

Struktura ldata przenosi opis blokady. Struktura flock jest zdefiniowana w <fcntl.h> i zawiera następujące składowe:

```
short l_type;           /* opisuje typ blokady */
short l_whence;         /* typ przesunięcia, jak dla lseek */
off_t l_start;          /* przesunięcie w bajtach */
off_t l_len;            /* wielkość segmentu w bajtach */
pid_t l_pid;            /* ustawiana przez polecenie F_GETLK */
```

Trzy składowe: l_whence, l_start i l_len określają segment pliku, który ma być zablokowany, testowany lub odblokowany. Składowa l_whence jest identyczna z trzecim argumentem lseek. Przybiera ona jedną z wartości: SEEK_SET, SEEK_CUR albo SEEK_END, aby wskazać, że przesunięcie ma być liczone od początku pliku, bieżącej pozycji wskaźnika odczytu-zapisu lub końca pliku. Składowa l_start podaje początkową pozycję segmentu pliku, względem miejsca wskazywanego przez l_whence. Składowa l_len jest długością segmentu pliku w bajtach; wartość równa zeru wskazuje segment od określonej pozycji początkowej do największego możliwego przesunięcia. Rysunek 8.1 objasnia te wartości dla przypadku, gdy składowa l_whence jest równa SEEK_CUR.



Rysunek 8.1 Parametry blokady

Składowa l_type podaje typ stosowanej blokady. Może ona przybierać jedną z trzech wartości, zdefiniowanych w <fcntl.h>:

- | | |
|---------|--|
| F_RDLCK | ma być zastosowana blokada odczytu |
| F_WRLCK | ma być zastosowana blokada zapisu |
| F_UNLCK | ma być usunięta blokada określonego segmentu |

Składowa l_pid jest istotna tylko wtedy, gdy wybrane polecenie fcntl to F_GETLK. Jeśli istnieje już blokada, która zabrania blokady opisanej przez inne składowe struktury, składowa l_pid będzie ustawiona na process-id procesu, który ją ustawił. Inne składowe struktury też będą ustawione odpowiednio przez system, dając więcej informacji o blokadzie utrzymywanej przez inny proces.

Ustawienie blokady za pomocą fcntl

Następujący przykład pokazuje, jak można zastosować fcntl do ustawienia blokady zapisu.

```
#include <unistd.h>
#include <fcntl.h>
.
.
.
struct flock my_lock;

my_lock.l_type = F_WRLCK;
my_lock.l_whence = SEEK_CUR;
my_lock.l_start = 0;
my_lock.l_len = 512;
```

```
fcntl(fd, F_SETLKW, &my_lock);
```

Ten przykład powinien zablokować 512 bajtów, począwszy od bieżącej pozycji wskaźnika odczytu-zapisu. Blokowana sekcja uchodzi za zarezerwowaną dla wyłącznego użytku procesu. Informacja o samej blokadzie jest umieszczana na wolnej pozycji w utrzymywanej przez system tablicy blokad.

Jeśli cała określona sekcja albo jej część jest już zablokowana przez inny proces, proces wywołujący zostanie wstrzymany (uśpiony), dopóki cała sekcja nie stanie się dostępna. Wstrzymany w ten sposób proces może być przerwany przez sygnał;

w szczególności może zostać wywołany alarm, aby określić czas oczekiwania. Jeśli wstrzymany proces nie zostanie przerwany, a sekcja ostatecznie stanie się wolna, blokada będzie zastosowana. Jeśli wystąpi błąd, kiedy na przykład do fcntl zostanie przekazany zły deskryptor pliku albo też systemowa tablica blokad jest pełna, wtedy zwracane jest -1.

Następny program przykładowy, lockit, otwiera plik locktest (który musi już istnieć) i blokuje jego pierwsze dziesięć bajtów za pomocą fcntl. Następnie rozwidla się; proces potomny próbuje zablokować pierwsze pięć bajtów; tymczasem proces rodzicielski jest wstrzymany na pięć sekund, a później wychodzi. Wtedy system automatycznie zwalnia blokadę założoną przez proces rodzicielski.

```
/* lockit -> demonstracja blokady za pomocą fcntl */

#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
    int fd;
    struct flock my_lock;

    /* ustaw parametry dla blokady zapisu */
    my_lock.l_type = F_WRLCK;
    my_lock.l_whence = SEEK_SET;
    my_lock.l_start = 0;
    my_lock.l_len = 10;

    /* otwórz plik */
    fd = open("locktest", O_RDWR);

    /* zablokuj pierwsze 10 bajtów */
    if( fcntl(fd, F_SETLK, &my_lock) == -1)
    {
        perror("parent: locking");
        exit(1);
    }

    printf("parent: locked record\n");

    switch(fork())
    {
        case -1:           /* błąd */
            perror("fork");
            exit(1);
        case 0:             /* potomek */
            my_lock.l_len = 5;
            if( fcntl(fd, F_SETLK, &my_lock) == -1)
            {
                perror("child: locking");
                exit(1);
            }
            printf("child: locked\n");
            printf("child: exiting\n");
            exit(0);
    }
}
```

```
sleep(5);

/* teraz wyjście, które zwalnia blokadę */
printf("parent: exiting\n");
exit(0);
}
```

Rzeczywiste wyjście tworzone przez lockit będzie wyglądać tak:

```
parent: locked record
parent: exiting
child: locked
child: exiting
```

Zwróci uwagę na kolejność, w jakiej są wyświetlane komunikaty. Pokazuje ona, że proces potomny nie może zastosować żądanej blokady, dopóki proces rodzicielski nie wyjdzie i nie zwolni swojej blokady, inaczej komunikat child: locked pojawiłby się jako drugi, a nie trzeci. Ten przykład pokazuje też, że blokada założona przez rodzica dotyczy potomka, nawet jeśli używane przez nich segmenty pliku nie są identyczne. Inaczej mówiąc, próba blokady zawiedzie, nawet jeśli sekcja tylko częściowo zachodzi na sekcję już zablokowaną. Program ilustruje kilka innych interesujących kwestii: po pierwsze, informacja o blokadzie nie jest dziedziczona przez wywołanie fork; w naszym przykładzie proces potomny i proces rodzicielski są niezależne, gdy chodzi o blokadę. Po drugie, wywołanie fcntl nie zmienia wskaźnika odczytu-zapisu dla pliku. Podczas wykonywania procesów: potomnego i rodzicielskiego, wskazuje on na początek pliku. Po trzecie, wszystkie blokady należące do procesu są usuwane automatycznie, gdy proces się kończy.

Odblokowanie za pomocą fcntl

Segment uprzednio zablokowany przez proces wywołujący może być odblokowany za pomocą ustawienia składowej l_type na F_UNLCK. Będzie to zwykle stosowane po upływie pewnego czasu od poprzedniego żądania fcntl. Jeśli jakiś inny proces czeka, aby zablokować sekcję, która została zwolniona, będzie ponownie uruchomiony.

Jeśli odblokowana sekcja znajduje się w środku wielkiej zablokowanej sekcji pliku, system utworzy dwie mniejsze blokady, które nie będą zawierały odblokowanej sekcji. Oznacza to, że zostanie zajęta dodatkowa pozycja w systemowej tablicy blokad. W konsekwencji żądanie odbezpieczenia może zanieść, nieco na przekór intuicji, jeśli systemowa tablica blokad jest pełna.

Na przykład w poprzednim programie lockit, proces rodzicielski zwalnia swoją blokadę przez wyjście z programu. Rodzic jednak mógłby jawnie zwolnić blokadę za pomocą następującego fragmentu kodu:

```
/* teraz rodzic zwalnia blokadę przed wyjściem */
printf("parent: unlocking\n");
my_lock.l_type = F_UNLCK;
if( fcntl(fd, F_SETLK, &my_lock) == -1)
{
    perror("parent: unlocking");
    exit(1);
}
```

Ponownie problem ACME Airlines

Teraz możemy dostarczyć rozwiążanie spornego problemu, na który natrafiliśmy w przykładzie ACME Airlines. Aby zapewnić integralność bazy danych, po prostu musimy następująco traktować krytyczną sekcję kodu w acmebook:

*zablokuj stosowny segment bazy danych za pomocą blokady zapisu
 aktualizuj segment bazy danych
 odblokuj segment bazy danych*

O ile żaden inny programu nie ominie mechanizmu blokady, żądanie blokady zapewnia, że wywołujący proces ma wyłączny dostęp do decydującej części bazy danych, kiedy tego potrzebuje. Żądanie odblokowania powoduje powtórne udostępnienie tego obszaru do publicznego użytku. Każda konkurująca kopia acmebook, próbująca dostępu do stosowej części bazy danych, gdy jest ona zablokowana, zostanie uśpiona, gdy tylko wykona własne żądanie blokady.

Rzeczywisty kod może wyglądać tak:

```
/* zarys procedury aktualizacji dla acmebook */

struct flock db_lock;
.

.

/* ustaw parametry blokady */
db_lock.l_type = F_WRLCK;
db_lock.l_whence = SEEK_SET;
db_lock.l_start = recstart;
db_lock.l_len = RECSIZE;
.

.

/* zablokuj rekord; uśpienie, jeśli */
/* rekord już jest zablokowany */
if( fcntl(fd, F_SETLKW, &db_lock) == -1)
    fatal("lock failed");

/* kod sprawdzania i aktualizacji danych rezerwacji */
.

.

/* teraz uwolnij rekord dla innych procesów */
db_lock.l_type = F_UNLCK;
fcntl(fd, F_SETLK, &db_lock);
```

Testowanie blokady

Jeśli wykonywana próba F_SETLK zawiedzie, ponieważ blokada jest już umieszczona, fcntl zwróci -1 i ustawi errno na EAGAIN lub EACCES (XSI określa obie wartości). Jeśli blokada istnieje, proces może określić za pomocą wywołania F_GETLK, który proces ustawił blokadę, na przykład:

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

```
#include <errno.h>
.

.

if( fcntl(fd, F_SETLK, &alock) == -1)
{
    if(errno == EACCES || errno == EAGAIN)
    {
        fcntl(fd, F_GETLK, &b_lock);
        fprintf(stderr, "record locked by %d\n", b_lock.l_pid);
    }
    else
        perror("unexpected lock error");
}
```

Zakleszczenie

Załóżmy, że dwa procesy, PA i PB, pracują na tym samym pliku. Przypuśćmy, że PA blokuje sekcję SX pliku, a PB blokuje oddzielną sekcję SY. Problem powstanie, jeśli PA spróbuje zablokować SY za pomocą F_SETLKW, a PB spróbuje zablokować SX za pomocą innego F_SETLK. Jeśli nic nie będzie zrobione, PA zostanie uśpiony, czekając aż PB zwolni SY, podczas gdy PB też zostanie uśpiony, czekając aż PA zwolni SX. O ile nie będzie interwencji z zewnątrz, te dwa procesy są skazane na pozostanie w stanie uśpienia, w śmiertelnym objęciu na zawsze.

Ten rodzaj sytuacji jest opisywany z oczywistych przyczyn jako **zakleszczenie** (ang. *deadlock*). Na szczęście Unix zabezpiecza przed taką sytuacją. Gdyby żąданie F_SETLKW spowodowało takie zakleszczenie, wywołanie zawiedzie – zwracane jest -1 i zmienna errno zostanie ustawiona na EDEADLK. Jednak fcntl może wykryć tylko zakleszczenie między dwoma procesami, a możliwe jest obmyślenie trójstronnego zakleszczenia. Złożone aplikacje, które używają blokad, aby uniknąć takiej sytuacji, powinny zawsze zawierać czas oczekiwania.

Następujący przykład wyjaśnia sprawę. W punkcie /*A*/ program blokuje bajty 0-9 pliku locktest. Następnie program rozwidla się. Proces potomny w punktach wskazanych przez komentarz /*B*/ i /*C*/ natychmiast blokuje bajty 10-14 i próbuje zablokować bajty 0-9. Ponieważ proces rodzicielski już je zablokował, proces potomny zostanie uśpiony. Tymczasem rodzic wykonuje 10-sekundowe wywołanie sleep. Jak dobrze pojedzie, pozwoli to procesowi potomnemu wykonać jego dwa wywołania blokujące. Kiedy rodzic się obudzi, spróbuje w punkcie /*D*/ zablokować bajty 10-14, które oczywiście zostały uprzednio zablokowane przez jego proces potomny. To jest punkt, w którym powinno wystąpić zakleszczenie i wywołanie fcntl powinno zawiść.

```
/* deadlock – demonstracja błędu zakleszczenia */

#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
    int fd;
```

```

struct flock first_lock;
struct flock second_lock;

first_lock.l_type = F_WRLCK;
first_lock.l_whence = SEEK_SET;
first_lock.l_start = 0;
first_lock.l_len = 10;

second_lock.l_type = F_WRLCK;
second_lock.l_whence = SEEK_SET;
second_lock.l_start = 10;
second_lock.l_len = 5;

fd = open("locktest", O_RDWR);

if( fcntl(fd, F_SETLKW, &first_lock) == -1)           /*A*/
    fatal("A");

printf("A: lock succeeded (proc %d)\n", getpid());

switch(fork()){
case -1:
    /*blad*/
    fatal("error on fork");
case 0:
    /*potomek*/
    if( fcntl(fd, F_SETLKW, &second_lock) == -1)      /*B*/
        fatal("B");
    printf("B: lock succeeded (proc %d)\n", getpid());
    if( fcntl(fd, F_SETLKW, &first_lock) == -1)          /*C*/
        fatal("C");
    printf("C: lock succeeded (proc %d)\n", getpid());
    exit(0);
default:
    /*rodzic*/
    printf("parent sleeping\n");
    sleep(10);
    if( fcntl(fd, F_SETLKW, &second_lock) == -1)      /*D*/
        fatal("D");
    printf("D: lock succeeded (proc %d)\n", getpid());
}

```

Kiedy ten program zostanie uruchomiony, utworzy następujące wyjście:

```

A: lock succeeded (proc 1410)
parent sleeping
B: lock succeeded (proc 1411)
D: deadlock situation detected/avoided
C: lock succeeded (proc 1411)

```

Blokada zawodzi tu w punkcie /*D*/ i perror drukuje odpowiedni komunikat systemu o błędzie. Zwróć uwagę, że gdy proces rodzicielski się zakończy i jego blokady zostaną zwolnione, proces potomny może wykonać swoją drugą blokadę.

W przykładzie wykorzystujemy procedurę o nazwie `fatal`, której używaliśmy w poprzednim rozdziale.

Ćwiczenie 8.1 Napisz procedurę powielającą działanie `read` i `write`, ale zawodzącą w przypadku istnienia blokady sekcji pliku. Przygotuj bliźniaczą procedurę `read`, blokującą sekcję, którą odczytuje (jeśli to możliwe). Każda blokada powinna być usuwana przy następnym wywołaniu procedury `read`.

Ćwiczenie 8.2 Opracuj i zaimplementuj ograniczony, zorientowany na rekordy schemat blokowania odczytu i zapisu, bazujący na numerze rekordu. (Wskazówka: możesz zablokować porcję pliku bliską maksymalnemu możliwemu przesunięciu w pliku, nawet jeśli nie istnieją tam dane. Poręca pliku w tej okolicy może być zarezerwowana, a każdy bajt może reprezentować rekord logiczny. Blokowanie może być wtedy używane dla celów znaczników).

8.3 Zaawansowane urządzenia IPC

8.3.1 Wprowadzenie i podstawowe pojęcia

Unix oferuje rozmaitość zaawansowanych mechanizmów komunikacji międzyprocesowej. Obecność tych zaawansowanych urządzeń IPC sprawia, że Unix jest systemem nadzwyczaj bogatym w obszarze komunikacji procesowej, a inwestor może użyć rozmaitych podejść przy programowaniu systemu wymagającego współpracujących zadań. Zaawansowane urządzenia IPC, które będziemy omawiać, należą do następujących kategorii:

1. przekazywanie komunikatów
2. semafory
3. wspólna pamięć.

Urządzenia te są powszechnie używane i pochodzą z Systemu V Uniksa. Powinieneś zwrócić uwagę, że alternatywy zostały zdefiniowane w ostatniej wersji POSIX-a.

Klucze urządzeń IPC

Interfejs programowy dla wszystkich trzech urządzeń IPC jest tak podobny, jak tylko to było możliwe, odbijając podobieństwo implementacji wewnątrz jądra. Najważniejszą wspólną cechą jest **klucz** (ang. *key*) urządzenia IPC. Klucze są liczbami używanymi do identyfikacji obiektów IPC w systemie Unix w podobny sposób, jak nazwa pliku identyfikuje plik. Inaczej mówiąc, klucz pozwala na wspólne użycie zasobów IPC przez kilka procesów. Identyfikowanym obiektem może być kolejka komunikatów, zestaw semaforów albo segment wspólnej pamięci. Rzeczywisty typ danych klucza jest określony przez zależny od implementacji typ `key_t`, zdefiniowany w systemowym pliku nagłówkowym `<sys/types.h>`.

Klucze nie są nazwami plików i mają mniejsze znaczenie. Powinny być wybierane uważnie (aby uniknąć kolizji między programami), by móc na podstawie numerów projektów przypisanych różnym projektantom na danym komputerze. (Jeden z dobrze znanych produktów baz danych używał szesnastkowej wartości klucza 0xDB: dobry pomysł, ale może przytrafić się i innym projektantom). Unix

dostarcza prostą funkcję biblioteczną, która odwzorowuje nazwę ścieżki pliku na klucz. Procedura ta nazywa się `ftok`.

Użycie

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

Procedura zwraca numer klucza w oparciu o informację powiązaną z path pliku. Parametr `id` też jest uwzględniany i dostarcza dodatkowego poziomu niepowtarzalności; innymi słowy, ta sama `path` będzie produkować różne klucze dla różnych wartości `id`. Procedura `ftok` jest raczej niezgrabna; na przykład jeśli plik zostanie usunięty, a następnie zastąpiony przez plik o tej samej nazwie, zwracane klucze powinny się różnić. Procedura zawodzi i zwraca (`key_t`) `-1`, jeśli `path` pliku nie istnieje. Procedura `ftok` jest prawdopodobnie najbardziej użyteczna w zastosowaniach, gdzie funkcje IPC są używane w manipulacji nazwanymi plikami lub kiedy dana jest nazwa pliku, będąca istotną, trwałą i niezmienną częścią aplikacji.

Operacja IPC get

Program używa klucza do tworzenia obiektu IPC lub wzmacnienia dostępu do istniejącego obiektu. Obie opcje są wywoływanie za pomocą operacji IPC `get`. Wynikiem operacji `get` jest całkowity identyfikator urządzenia (ang. *facility identifier*) IPC, który może być używany w wywołaniu innych procedur IPC. Trzymając się dalej analogii pliku powiemy, że operacja `get` jest podobna do wywołania `creat` lub `open`, a identyfikator urządzenia IPC działa trochę jak deskryptor pliku. W rzeczywistości, w odróżnieniu od deskryptorów pliku, identyfikator urządzenia IPC jest unikatowy. Różne procesy muszą używać tej samej wartości dla tego samego obiektu IPC.

Jako przykład, przyjrzyjmy się, jak następująca instrukcja używa procedury IPC `msgget` do utworzenia nowej kolejki komunikatów (nie zastanawiaj się, czym faktycznie jest kolejka komunikatów; omówimy to później):

```
mqid = msgget( (key_t)0100, 0644 | IPC_CREAT | IPC_EXCL );
```

Pierwszy argument `msgget` to klucz kolejki komunikatów. Jeśli wywołanie jest pomyślne, procedura zwraca w `mqid` wartość nieujemną, która działa jako identyfikator kolejki komunikatów. Odpowiadającymi procedurami dla semaforów i wspólnej pamięci są odpowiednio `semget` i `shmget`.

Inne operacje IPC

Istnieją jeszcze dwa inne typy operacji, które mogą być wykonywane z urządzeniami IPC. Po pierwsze, są to operacje kontrolne, które mogą być używane do uzyskania informacji o stanie lub do ustawiania wartości kontrolnych. Rzeczywiście procedurami wykonującymi te funkcje są `msgctl`, `semctl` i `shmctl`. Po drugie, istnieją bardziej specyficzne operacje zgrupowane pod nagłówkiem ope-

racji IPC, które wykonują interesującą pracę. Dla każdego urządzenia dostępne są różne operacje i będą one omawiane w odpowiednich podrozdziałach. Na przykład istnieją dwie operacje komunikatów: `msgsnd` umieszczająca komunikat w kolejce komunikatów, i `msgrcv` odczytująca komunikat z kolejki komunikatów.

Struktura danych stanu

Kiedy obiekt IPC jest tworzony, system tworzy też strukturę stanu urządzenia IPC (ang. *IPC facility status structure*), zawierającą każdą informację administracyjną powiązaną z obiektem. Istnieje jeden typ struktury stanu dla komunikatów, semaforów i wspólnej pamięci. Każdy typ zawiera też informację, która wiąże się z określonym urządzeniem IPC. Jednak wszystkie trzy typy struktur stanu zawierają wspólną strukturę uprawnień, identyfikowaną jako `ipc_perm`. Zawiera ona następujące składowe:

```
uid_t cuid;           /* user-id twórcy obiektu IPC */
gid_t cgid;           /* group-id twórcy */
uid_t uid;            /* efektywny user-id */
gid_t gid;            /* efektywny group-id */
mode_t umode;         /* uprawnienia */
```

Struktura ta decyduje, czy użytkownik może odczytać obiekt IPC (czyli otrzymać informację o obiekcie) lub zapisać do niego (co oznacza manipulację nim). Uprawnienia są budowane dokładnie w taki sam sposób, jak dla plików. Tak więc wartość 0644 dla składowej `umode` oznacza, że właściciel może odczytywać i zapisywać powiązany obiekt, podczas gdy inni użytkownicy mogą go tylko odczytać. Zauważ, że są to efektywne identyfikatory użytkownika i grupy (zarejestrowane w składowych `uid` i `gid`), które w połączeniu z `umode` określają prawa dostępu. Powinno być także jasne, że prawo wykonywania nie ma tu znaczenia. Jak zwykle, super-użytkownik ma wolną rękę. W przeciwnieństwie do innych konstrukcji Uniksa, gdy tworzone jest urządzenie IPC, wartość `umask` użytkownika nie ma znaczenia.

8.3.2 Przekazywanie komunikatów

Zaczniemy nasze szczegółowe badanie urządzeń IPC od przyjrzenia się funkcjom pierwotnym przekazywania komunikatów.

W istocie, komunikat jest po prostu sekwencją znaków lub bajtów (niekoniecznie zakończonych zerem). Komunikaty są przekazywane między procesami za pomocą kolejek komunikatów (ang. *message queues*), tworzonych lub udostępnianych przez funkcję pierwotną `msgget`. Gdy kolejka jest utworzona, proces, mając stosowne uprawnienia kolejki, może umieścić w niej komunikat za pomocą `msgsnd`. Inny proces może wtedy odczytać ten komunikat za pomocą funkcji `msgrcv`, która też go usuwa z kolejki. Z tego krótkiego opisu wynika, że przekazywanie komunikatów w sensie IPC jest podobne do tego, co można było osiągnąć za pomocą odczytów i zapisów do potoku (jak to omawialiśmy w podrozdziale 7.1.2).

Funkcja `msgget` jest zdefiniowana następująco:

Użycie

```
#include <sys/msg.h>
int msgget(key_t key, int permflags);
```

Najbliższą analogią tego wywołania jest równolegle działanie funkcji `open` lub `creat`. Jak widzieliśmy w podrozdziale 8.3.1, parametr `key`, zasadniczo tylko numer, identyfikuje kolejkę komunikatów w systemie. Jeśli wywołanie jest pomyślne i utworzona zostanie nowa kolejka lub udostępniona istniejąca kolejka, funkcja `msgget` powinna zwrócić **identyfikator kolejki komunikatów** (ang. *message queue identifier*) w postaci nieujemnej liczby całkowitej.

Parametr `permflags` określa dokładnie działanie wykonywane przez `msgget`. Mogą tu pojawić się dwie stale, zdefiniowane w pliku `<sys/ipc.h>`; używane samodzielnie albo połączone operacją alternatywy bitowej (OR):

`IPC_CREAT` Ta stała mówi, że funkcja `msgget` ma utworzyć kolejkę komunikatów dla wartości `key`, o ile ona jeszcze nie istnieje. Trzymając się naszej metafory pliku powiemy, że ten znacznik powoduje odgrywanie przez `msgget` roli wywołania `creat`, chociaż kolejka komunikatów nie będzie nadpisana, jeśli już istnieje. Gdy znacznik `IPC_CREAT` nie jest ustawiony, dopóki istnieje kolejka dla tego klucza, `msgget` zwraca istniejący identyfikator kolejki.

`IPC_EXCL` Jeśli ten znacznik i znacznik `IPC_CREAT` zostały ustawione, wtedy wywołanie jest przeznaczone tylko do utworzenia kolejki komunikatów. Tak więc, gdy kolejka dla `key` już istnieje, wywołanie `msgget` zawiedzie i zwróci -1. Zmienna błędu `errno` zawiera wtedy wartość `EEXIST`.

Kiedy tworzona jest kolejka komunikatów, do nadania jej uprawnień używane jest dziewięć mniej znaczących bitów `permflags`, prawie jak tryb pliku. Są one pamiętane w strukturze `ipc_perm`, tworzonej razem z samą kolejką.

Teraz możemy wrócić do przykładu z podrozdziału 8.3.1:

```
mqid = msgget( (key_t)0100, 0644 | IPC_CREAT | IPC_EXCL);
```

To wywołanie zostało przeznaczone do utworzenia (i tylko utworzenia) kolejki komunikatów dla wartości klucza (`key_t`) 100. Jeśli wywołanie jest pomyślne, kolejka będzie miała uprawnienia 0644. Są one interpretowane w ten sam sposób, jak uprawnienia pliku, wskazując, że twórca kolejki może wysyłać lub odczytywać komunikaty z kolejki, podczas gdy członkowie grupy twórcy i wszyscy inni mogą z niej tylko odczytywać. Jeśli to konieczne, można później użyć `msgctl` do zmiany powiązanych z kolejką uprawnień i praw własności.

Operacje kolejki komunikatów: `msgsnd` i `msgrcv`

Gdy kolejka została już uruchomiona, do manipulacji nią można używać następujących dwóch operacji pierwotnych:

Użycie

```
#include <sys/msg.h>
int msgsnd(int mqid, const void *message, size_t size,
           int flags);
int msgrcv(int mqid, void *message, size_t size, long msg_type,
           int flags);
```

Pierwsza z nich, `msgsnd`, służy do dodawania komunikatu do kolejki wskazywanej przez parametr `mqid`, którego wartość uzyskuje się z wywołania `msgget`.

Sam komunikat jest zawarty, co nie powinno stanowić zaskoczenia, w strukturze `message`, zadeklarowanej w dostarczanym przez użytkownika wzorcu. Forma tego wzorca jest następująca:

```
struct mymsg {
    long mtype;           /* typ komunikatu */
    char mtext[SOMEVALUE]; /* tekst komunikatu */
};
```

Składowa `mtype` może być używana przez programistę do klasyfikacji komunikatów, z każdą możliwą wartością reprezentującą inną potencjalną kategorię. Znaczące są tu tylko wartości dodatnie; wartości ujemne i zero nie będą działać. Składowa `mtext` zawiera tekst samego komunikatu (stała `SOMEVALUE` jest całkowicie dowolna). Długość aktualnie wysyłanego komunikatu została podana w parametrze `size` funkcji `msgsnd` i może zawierać się w przedziale od zera do wartości mniejszej niż `SOMEVALUE` albo maksimum określonego przez system.

Parametr `flags` dla `msgsnd` może przybrać tylko jedną znaczącą wartość: `IPC_NOWAIT`. Jeśli wartość `IPC_NOWAIT` nie jest ustawiona, proces wywołujący będzie wstrzymany, o ile brak jest dostatecznych zasobów systemowych do wysłania komunikatu. W praktyce może się to zdarzyć, kiedy całkowita długość komunikatów w kolejce przekroczy maksimum przypadające na kolejkę lub maksimum systemowe. Jeśli wartość `IPC_NOWAIT` jest ustawiona, to gdy komunikat nie może być wysłany, wywołanie natychmiast powróci. Zwracaną wartością będzie wtedy -1, a zmienna `errno` będzie ustawiona na `EAGAIN`, oznaczając konieczność ponownej próby.

Funkcja `msgrcv` może też zanieść z powodu uprawnień związanych z kolejką komunikatów. Na przykład jeśli użytkownik nie ma żadnego efektywnego user-id ani efektywnego group-id związanego z kolejką, a prawa dostępu kolejki wynoszą 0660, wtedy wywołanie `msgsnd` dla tej kolejki zawiedzie. Zmienna błędu `errno` będzie wtedy zawierać `EACCES`.

Przejdźmy teraz do odczytu komunikatów. Funkcja `msgrecv` jest używana do odczytu komunikatu z kolejki identyfikowanej przez `mqid`, pod warunkiem, że prawa dostępu kolejki pozwalają procesowi to robić. Odczytanie komunikatu powoduje usunięcie go z kolejki.

Tym razem parametr `message` jest używany do trzymania odebranego komunikatu, a parametr `size` podaje maksymalną długość, która może być trzymana wewnątrz struktury. Jeśli wywołanie jest pomyślne, wtedy `retval` zawiera długość otrzymanego komunikatu.

Parametr `msg_type` określa dokładnie, jaki komunikat został właściwie odebrany. Przeprowadza on selekcję stosownie do wartości pola `mtype` komunikatu. Jeśli parametr `msg_type` jest równy zeru, odczytywany będzie pierwszy komunikat z kolejki, to znaczy najwcześniej wysłany. Jeśli `msg_type` ma różną od zera, dodatnią wartość, wtedy odczytywany będzie pierwszy komunikat z tą wartością. Na przykład, jeśli kolejka zawiera komunikaty z wartościami `mtype` 999, 5 i 1, funkcja `msgrecv` jest wywoływana z parametrem `msg_type` ustawionym na 5, wtedy odczytywany będzie komunikat typu 5. Na koniec, jeśli `msg_type` ma różną od zera, ujemną wartość, odczytywany będzie pierwszy komunikat z najniższą liczbą `mtype`, mniejszą niż albo równą wartości bezwzględnej `msg_type`. To samo można powiedzieć prościej – posłuży nam do tego nasz poprzedni przykład. Mamy tam w kolejce trzy komunikaty z wartościami `mtype` 999, 5 i 1. Jeśli parametr `msg_type` jest ustawiony na -999 (oczywiście rzutowanym na `long`), a funkcja `msgrecv` wywołana trzy razy, wtedy komunikaty będą odebrane w kolejności 1, 5 i 999.

Ostatni parametr, `flags`, ponownie zawiera informację kontrolną. Dwie wartości: `IPC_NOWAIT` i `MSG_NOERROR`, mogą być ustawiane samodzielnie lub sumowane za pomocą alternatywy bitowej (OR). Wartość `IPC_NOWAIT` oznacza to samo, co przedtem; jeśli nie jest ustawiona, wtedy proces będzie uspiony, jeśli nie ma odpowiedniego komunikatu w kolejce, wracając, kiedy nadziejdie komunikat stosownego typu. Jeśli jest ustawiona, wywołanie powróci natychmiast, bez wzgledu na okoliczności.

Gdy komunikat jest dłuższy niż `size` bajtów, to jeśli wartość `MSG_NOERROR` została ustawiona, komunikat będzie obcięty; w przeciwnym przypadku wywołanie `msgrecv` zawiedzie. Niestety, nie ma sposobu dowiedzenia się, że nastąpiło obcięcie. Ten podrozdział jest dość trudny; formuły urządzeń IPC wydają się raczej przeciwstawne Uniksowi z racji swojej złożoności i stylu. Jednak procedury przekazywania komunikatów są w rzeczywistości proste w użyciu, z wieloma potencjalnymi zastosowaniami. Mamy nadzieję, że dowiedzie tego następny przykład.

Przykład przekazywania komunikatów: kolejka priorytetowa

W tym podrozdziale będziemy rozwijać prostą aplikację przekazywania komunikatów. Celem jest implementacja systemu kolejki, w którym każdy element ma nadany priorytet. Proces serwera powinien pobierać elementy z kolejki i przetwarzać je w jakiś sposób. Elementy znajdujące się w kolejce mogą być na przykład nazwami plików, a proces serwera może kopiować je na drukarkę. Jest to podobne do przykładu FIFO z podrozdziału 7.2.1.

Zaczynamy od pliku nagłówkowego o nazwie `q.h`, który wygląda następująco:

```
/* q.h -- nagłówek dla przykładowego urządzenia komunikatów */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <errno.h>

#define QKEY      (key_t)0105 /* identyfikuje key dla kolejki */
#define QPERM     0660        /* prawa dostępu dla kolejki */
#define MAXOBN    50          /* maksymalna długość nazwy obiektu */
#define MAXPRIOR 10          /* maksymalny poziom priorytetu */

struct q_entry {
    long mtype;
    char mtext[MAXOBN+1];
};
```

Pierwsza część tego pliku grupuje dyrektywy `#include`. Definicja `QKEY` podaje wartość klucza, który będzie identyfikował kolejkę komunikatów w systemie. `QPERM` podaje związane z kolejką prawa dostępu. Ponieważ są one zdefiniowane jako 0660, właściciel kolejki i członkowie jej grupy będą mogli odczytywać i zapisywać kolejkę komunikatów. Jak zobaczymy, `MAXOBN` i `MAXPRIOR` narzucają ograniczenie komunikatowi, który może być umieszczony w kolejce. Końcowa część włączanego pliku zawiera definicję wzorca struktury z wyróżnikiem `q_entry`. Będziemy używać struktur tego typu do trzymania komunikatów nadawanych i odbieranych przez nasze procedury.

Najpierw omówimy procedurę o nazwie `enter`, która umieszcza zakończoną zerem nazwę obiektu w kolejce. Ma ona następującą formę:

```
/* enter -- umieszcza obiekt w kolejce */

#include "q.h"

int enter(char *objname, int priority)
{
    int len, s_qid;
    struct q_entry s_entry; /* struktura do trzymania komunikatów */

    /* zatwierdź długość nazwy, poziom priorytetu */

    if( (len = strlen(objname)) > MAXOBN )
    {
        warn("name too long");
        return (-1);
    }
    if(priority > MAXPRIOR || priority < 0)
    {
        warn("invalid priority level");
        return (-1);
    }

    /* zainicjuj, jeśli to konieczne, kolejkę komunikatów */
```

```

if( (s_qid = init_queue()) == -1)
    return (-1);

/* zainicjuj s_entry */
s_entry.mtype = (long)priority;
strncpy(s_entry.mtext, objname, MAXOBN);

/* wyslij komunikat, czekajac jezeli trzeba */

if(msgsnd(s_qid, &s_entry, len, 0) == -1)
{
    perror("msgsnd failed");
    return (-1);
}
else
    return (0);
}

```

Pierwsze działanie procedury `enter` to sprawdzenie długości nazwy obiektu i poziomu priorytetu. Zwróć uwagę, że minimalną wartością priorytetu jest 1, ponieważ wartość zerowa spowodowałaby niepowodzenie wywołania `msgsnd`. Następnie procedura `enter` otwiera kolejkę za pomocą wywołania `init_queue`. Niebawem zobaczymy, jak jest ono implementowane.

Jeśli wszystko jest w porządku, procedura buduje komunikat i próbuje wysłać go za pomocą `msgsnd`. Zwróć uwagę, jak używamy struktury `q_entry` o nazwie `s_entry` do przechowywania komunikatów. Zauważ, że ostatni parametr `msgsnd` jest zerowy. Oznacza to, że system będzie wstrzymywał proces wywołujący, jeśli kolejka jest pełna (ponieważ nie został ustawiony znacznik `IPC_NOWAIT`).

Procedura `enter` wskazuje problemy za pomocą użycia `warn` lub funkcji bibliotecznej `perror`. Dla uproszczenia zaimplementowaliśmy `warn` następująco:

```
#include <stdio.h>

int warn(char *s)
{
    sprintf(stderr, "warning: %s\n", s);
}
```

Wrzeciwistym systemie byłoby lepiej, gdyby procedura `warn` zapisywała problemy do pliku dziennika błędów.

Wracając do `init_queue`, powiemy, że cel tej funkcji jest jasny. Inicjuje ona, jeśli to możliwe, identyfikator kolejki komunikatów, a w przeciwnym przypadku zwraca już związany z kolejką identyfikator kolejki komunikatów.

```
/* init_queue -- pobiera identyfikator kolejki */

#include "q.h"

int init_queue(void)
{
    int queue_id;

    /* spróbuj utworzyć lub otworzyć kolejkę komunikatów */
    if( (queue_id = msgget(QKEY, IPC_CREAT | QPERM)) == -1)
```

```

        perror("msgget failed");

        return (queue_id);
}

Nastepna procedura jest uzywana przez proces serwera do obslugi elementow
kolejki. Nadalismy jej pomyslowa nazwe serve. Stanowi ona sensownie prostą
odwrotnosc procedury enter.

/* serve -- obsluguje obiekty kolejki z najwyzszym priorytetem */

#include "q.h"

int serve(void)
{
    int mlen, r_qid;
    struct q_entry r_entry;

    /* zainicjuj w mire potrzeb kolejke komunikatow */

    if((r_qid = init_queue()) == -1)
        return (-1);

    /* pobierz i przetwierz nastepny komunikat, czekajac w mire potrzeb */

    for(;;)
    {
        if((mlen = msgrcv(r_qid, &r_entry, MAXOBN,
                           (-1 * MAXPRIOR), MSG_NOERROR)) == -1)
        {
            perror("msgrcv failed");
            return (-1);
        }
        else
        {
            /* zapewnij, ze jest to napis */
            r_entry.mtext[mlen] = '\0';

            /* nazwa obiektu procesu */
            proc_obj(&r_entry);
        }
    }
}
```

Zwróć uwagę, jak wywoływana jest funkcja `msgrcv`. Ponieważ jako parametr podana jest wartość ujemna (`-1 * MAXPRIOR`), system najpierw sprawdzi w kolejce obecność komunikatów z `mtype` równym 1, następnie równym 2, i tak dalej, aż do i włącznie z `MAXPRIOR`. Innymi słowy, najwyższy priorytet mają komunikaty z najniższymi numerami. Procedura `proc_obj` rzeczywiście działa. W przypadku systemu wydruku może ona kopiować pliki na drukarkę.

Następujące dwa proste programy demonstrują współdziałanie tych procedur: `etest` umieszcza element w kolejce, podczas gdy `stest` przetwarza element (w rzeczywistości tylko drukuje zawartość i typ komunikatu).

Program etest

```
/* etest -- wprowadza nazwy obiektów do kolejki */
#include <stdio.h>
#include <stdlib.h>
#include "q.h"

main(int argc, char **argv)
{
    int priority;

    if(argc != 3)
    {
        fprintf(stderr, "usage: %s objname priority\n", argv[0]);
        exit(1);
    }

    if((priority = atoi(argv[2])) <= 0 || priority > MAXPRIOR)
    {
        warn("invalid priority");
        exit(2);
    }

    if(enter(argv[1], priority) < 0)
    {
        warn("enter failure");
        exit(3);
    }

    exit(0);
}
```

Program stest

```
/* stest -- prosty serwer kolejki */
#include <stdio.h>
#include "q.h"

main()
{
    pid_t pid;

    switch(pid = fork()){
    case 0: /* potomek */
        serve();
        break; /* w rzeczywistości serwer nigdy nie wyjdzie */
    case -1:
        warn("fork to start server failed");
        break;
    default:
        printf("server process pid is %d\n", pid);
    }
    exit(pid != -1 ? 0 : 1);
}

int proc_obj(struct q_entry *msg)
```

```
{
    printf("\npriority: %d name: %s\n", msg->mtype, msg->mtext);
}
```

Oto przykład użycia tych dwóch prostych programów. Do kolejki zostały wprowadzone za pomocą *etest* cztery komunikaty, zanim został uruchomiony serwer *stest*. Zwróć uwagę na kolejność, w jakiej są drukowane komunikaty:

```
$ etest objname1 3
$ etest objname2 4
$ etest objname3 1
$ etest objname4 9
$ stest
server process pid is 2545
priority: 1 name: objname3
priority: 3 name: objname1
priority: 4 name: objname2
priority: 9 name: objname4
```

Ćwiczenie 8.3 Dostosuj programy *enter* i *serve* tak, aby do serwera wysyłane były komunikaty kontrolne. Zarezerwuj dla nich jeden typ komunikatów (jaka powinna być wartość priorytetu?). Zaimplementuj następujące opcje:

1. zatrzymanie serwera
2. opróżnienie kolejki z wszystkich komunikatów
3. opróżnienie z komunikatów o danym priorytecie.

Funkcja systemowa msgctl

Procedura *msgctl* służy trzem celom: pozwala procesowi uzyskać informację o stanie kolejki komunikatów, zmienić niektóre ze związków z kolejką ograniczeń lub usunąć całkowicie kolejkę z systemu.

Użycie

```
#include <sys/msg.h>
int msgctl(int mqid, int command, struct msqid_ds *msq_stat);
```

Parametr *mqid* jest oczywiście ważnym identyfikatorem kolejki komunikatów. Pomijany chwilowo parametr *command*. Trzeci parametr *msq_stat* zawiera adres struktury *msqid_ds*. Ta nieatrakcyjnie nazwana struktura jest zdefiniowana w *<sys/msg.h>* i zawiera następujące składowe:

```
struct ipc_perm msg_perm; /* prawa własności/dostępu */
msgqnum_t msg_qnum; /* liczba komunikatów w kolejce */
msglen_t msg_qbytes; /* maksymalna liczba bajtów kolejki */
```

```

pid_t msg_lspid; /* pid ostatniej funkcji msgsnd */
pid_t msg_lrid; /* pid ostatniej funkcji msgrcv */
msg_stime; /* czas ostatniej funkcji msgsnd */
msg_rtime; /* czas ostatniej funkcji msgrcv */
time_t msg_ctime; /* czas ostatniej zmiany */

```

Strukturę `ipc_perm` spotkaliśmy już przedtem. Zawiera ona prawa własności i prawa dostępu związane z kolejką komunikatów. Typy `msgqnum_t`, `msglen_t`, `pid_t` i `time_t` są zależne od systemu. Zmienna `time_t` zawiera liczbę sekund, jakie minęły od godziny 00:00 GMT 1 stycznia 1970 roku. (Następny przykład pokaże, jak zamienić takie wartości na czytelne napisy).

Parametr `command` funkcji `msgctl` mówi systemowi, jaka operacja ma być wykonyana. Istnieją tu trzy dostępne opcje, dotyczące wszystkich trzech urządzeń IPC. Są one identyfikowane za pomocą stałych zdefiniowanych w `<sys/ipc.h>`.

`IPC_STAT` Każe systemowi umieścić w `msq_stat` informację o stanie struktury

`IPC_SET` Używana jest do ustawienia wartości zmiennych kontrolnych dla kolejki komunikatów, stosownie do informacji zawartych w `msq_stat`. Zmieniane mogą być tylko następujące pozycje:

```

msq_stat.msg_perm.uid
msq_stat.msg_perm.gid
msq_stat.msg_perm.mode
msq_stat.msg_qbytes

```

Operacja `IPC_SET` zakończy się pomyślnie tylko wtedy, jeśli jest wykonywana przez super-użytkownika lub bieżącego właściciela kolejki wskazywanego przez `msq_stat.msg_perm.uid`. W dodatku tylko super-użytkownik może powiększyć ograniczenie `msg_qbytes`, podające maksymalną liczbę znaków, które mogą znajdować się jednocześnie w kolejce.

`IPC_RMID` Usuwa kolejkę komunikatów z systemu. I znów może to być zrobione tylko przez super-użytkownika lub właściciela kolejki. Jeśli parametr `command` jest ustawiony na `IPC_RMID`, to `msq_stat` będzie ustawiony na `NULL`.

Następujący przykładowy program `show_msg` drukuje kilka informacji stanu związanych z kolejką komunikatów. W zamierzeniu powinien być wywoływany następująco:

```
$ show_msg keyvalue
```

Program `show_msg` używa procedury bibliotecznej `ctime` do zamiany wartości `time_t` na postać czytelną. (Procedura `ctime` i procedury jej towarzyszące będą omawiane w rozdziale 12). Tekst programu wygląda następująco:

```
/* showmsg -- pokazuje szczegóły kolejki komunikatów */
```

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>

```

```

#include <time.h>

void mqstat_print(key_t, int, struct msqid_ds *);

main(int argc, char **argv)
{
    key_t mkey;
    int msq_id;
    struct msqid_ds msq_status;

    if(argc != 2)
    {
        fprintf(stderr, "usage: showmsg keyval\n");
        exit(1);
    }

    /* pobierz identyfikator kolejki komunikatów */
    mkey = (key_t)atoi(argv[1]);
    if((msq_id = msgget(mkey, 0)) == -1)
    {
        perror("msgget failed");
        exit(2);
    }

    /* pobierz informacje o stanie */
    if(msgctl(msq_id, IPC_STAT, &msq_status) == -1)
    {
        perror("msgctl failed ");
        exit(3);
    }

    /* drukuj informację o stanie */
    mqstat_print(mkey, msq_id, &msq_status);
    exit(0);
}

void mqstat_print(key_t mkey, int mqid, struct msqid_ds *mstat)
{
    printf("\nKey %d, msg_qid %d\n\n", mkey, mqid);

    printf("%d message(s) on queue\n\n", mstat->msg_qnum);

    printf("Last send by proc %d at %s\n", mstat->msg_lspid,
          ctime(&(mstat->msg_stime)));
    printf("Last recv by proc %d at %s\n", mstat->msg_lrid,
          ctime(&(mstat->msg_rtime)));
}

```

Ćwiczenie 8.4 Dostosuj program `show_msg` tak, aby drukował związane z kolejką komunikatów prawa własności i dostępu.

Ćwiczenie 8.5 Napisz program `msg_chmod`, który zmienia prawa związane z kolejką komunikatów. Wzoruj się na programie `chmod`. I znów, kolejka komunikatów powinna być identyfikowana przez wartość jej klucza.

8.3.3 Semafora

Semafor jako konstrukcja teoretyczna

Pojęcie semafora zostało po raz pierwszy sformułowane przez holenderskiego teoretyka E.W.Dijkstrę jako rozwiązanie problemów synchronizacji procesów. W tej terminologii semafor *sem* może być traktowany jak zmienna całkowita, na której dozwolone są następujące operacje (zauważ, że mnemoniki *p* i *v* pochodzą od holenderskiej terminologii dla *wait* i *signal*; ta druga nie powinna być mieszana ze starą funkcją Uniksa *signal*):

p(sem) lub *wait(sem)*

```
if (sem != 0)
    zmniejsz sem o jeden
else
    czekaj, aż sem stanie się różne od zera, wtedy zmniejsz
    v(sem) lub signal(sem)
```

```
zwiększ sem o jeden
if (kolejka czekających procesów nie jest pusta)
    uruchom ponownie pierwszy proces z kolejki oczekujących
```

Część testująca i ustawiająca obu operacji musi być niepodzielna, co oznacza, że tylko jeden proces może w danym czasie zmienić wartość *sem*.

Cieszy, że przy semaforach warunek:

```
(początkowa wartość semafora
+ liczba operacji v
- liczba zakończonych operacji p) >= 0
```

jest zawsze prawdziwy. Jest to **niezmiennik semafora** (ang. *semaphore invariant*). Naukowcy z zakresu informatyki lubią takie warunki niezmienników, ponieważ dzięki nim programy są podatne na systematyczne, rygorystyczne dowody.

Semafora mogą być używane w rozmaity sposób. Najprościej – do zapewnienia **wzajemnego wykluczania** (ang. *mutual exclusion*), gdzie tylko jeden proces może wykonywać w danym czasie określony obszar kodu. Rozważmy następujący zarys programu:

p(sem);

coś ciekawego...

v(sem);

Załóżmy dalej, że początkowa wartość *sem* wynosi jeden. Na podstawie niezmiennika semafora możemy zobaczyć, że:

(liczba zakończonych operacji p – liczba zakończonych operacji v) <= początkowa wartość semafora

lub:

(liczba zakończonych operacji p – liczba zakończonych operacji v) <= 1

Innymi słowy, w danym czasie tylko jeden proces może wykonywać grupę instrukcji między tymi szczególnymi operacjami *p* i *v*. Taki obszar programu jest często nazywany sekcją krytyczną (ang. *critical section*).

Na tych założeniach oparta jest uniksowa implementacja semaforów, chociaż rzeczywiste oferowane urządzenia są raczej bardziej ogólne (i być może zbyt złożone). Pierwsze procedury, które będziemy omawiać, to *semget* i *semctl*.

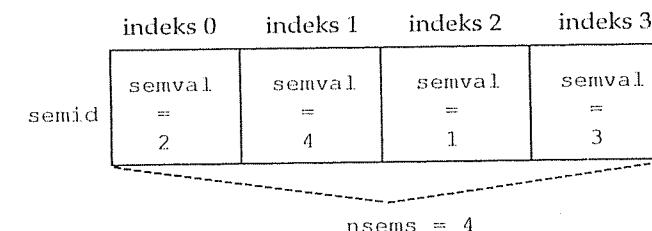
Funkcja systemowa *semget*

Użycie

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int permflags);
```

Funkcja *semget* jest analogiczna do funkcji *msgget*. Dodatkowy parametr *nsems* podaje liczbę semaforów wymaganych w zestawie semaforów; to ważne zagadnienie – operacje semaforowe Uniksa są nastawione na pracę z zestawami semaforów, a nie pojedynczymi obiektami. Rysunek 8.2 pokazuje zestaw semaforów. Jak zobaczymy, komplikuje to interfejs pozostałych procedur semaforów.

Wartością zwracaną przez pomyślne wywołanie *semget* jest **identyfikator zestawu semaforów** (ang. *semaphore set identifier*). Został on przedstawiony na rysunku 8.2 za pomocą *semid*. Jak zwykle w C, indeks zestawu semaforów ma zakres od 0 do *nsems*-1.



Rysunek 8.2 Zestaw semaforów

Z każdym semaforem w zestawie związane są następujące wartości:

semval

Wartość semafor, zawsze dodatnia liczba całkowita. Musi być ustawiana za pomocą funkcji systemowej *semset*; oznacza to, że semafor nie jest bezpośrednio dostępny dla programu jako obiekt danych.

<i>semid</i>	Identyfikator procesu, który ostatnio miał do czynienia z semaforem.
<i>semcnt</i>	Liczba procesów, które czekają, aż semafor osiągnie wartość większą niż jego aktualna wartość.
<i>semzcnt</i>	Liczba procesów, które czekają, aż semafor osiągnie wartość zerową.

Funkcja systemowa semctl

Użycie

```
#include <sys/sem.h>
int semctl(int semid, int sem_num, int command,
union semun ctl_arg);
```

Jak to wynika z definicji, funkcja `semctl` jest znacznie bardziej skomplikowana niż funkcja `msgctl`. Parametr `semid` musi być ważnym identyfikatorem semafora, zwracanym przez wywołanie funkcji `semget`. Parametr `command` ma podobne znaczenie jak w funkcji `msgctl`, podając dokładnie wymaganą funkcję. Funkcja ta należy do trzech kategorii: standardowe funkcje IPC (takie jak `IPC_STAT`), funkcje działające tylko na pojedynczy semafor i funkcje dotyczące całego zestawu semaforów. Wszystkie dostępne funkcje są pokazane w tabeli 8.1.

Tabela 8.1 Kody funkcji `semctl`

Standardowe funkcje IPC

(zauważ, że struktura `semid_ds` jest zdefiniowana w `<sys/sem.h>`).

<code>IPC_STAT</code>	umieszcza informację o stanie w <code>ctl_arg.stat</code>
<code>IPC_SET</code>	ustawia informację o prawach własności i dostępu z <code>ctl_arg.stat</code>
<code>IPC_RMID</code>	usuwa zestaw semaforów z systemu

Operacje na pojedynczym semaforze

(dotyczą semafora `sem_num`, wartości zwracanej przez `semctl`)

<code>GETVAL</code>	zwraca wartość semafora (to znaczy <code>semval</code>)
<code>SETVAL</code>	ustawia wartość semafora w <code>ctl_arg.val</code>
<code>GETPID</code>	zwraca wartość <code>semid</code>
<code>GETNCNT</code>	zwraca <code>semcnt</code> (zobacz powyżej)
<code>GETZCNT</code>	zwraca <code>semzcnt</code> (zobacz powyżej)

Operacje na wszystkich semaforach

<code>GETALL</code>	umieszcza wszystkie <code>semvals</code> w <code>ctl_arg.array</code>
<code>SETALL</code>	ustawia wszystkie <code>semvals</code> zgodnie z <code>ctl_arg.array</code>

Parametr `sem_num` jest używany z drugą grupą opcji `semctl` do identyfikacji konkretnego semafora. Końcowy parametr `ctl_arg` jest unią zdefiniowaną następująco:

```
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
```

Każda składowa unii reprezentuje różnych typ wartości ustawianych dla każdego z trzech typów funkcji `semctl`. Na przykład jeśli `semval` jest równy `SETVAL`, będzie użyta wartość `ctl_arg.val`.

Jednym z ważniejszych zastosowań `semctl` jest ustawienie początkowej wartości semafora, ponieważ `semget` nie pozwala tego zrobić procesowi. Następująca przykładowa funkcja może być używana przez program do tworzenia pojedynczego semafora albo po prostu do otrzymania związanego z nim identyfikatora zestawu semaforów. Jeśli semafor jest naprawdę tworzony, zostaje mu przypisana początkowa wartość jednego z `semctl`.

```
/* initsem -- inicjacja semafora */

#include "pv.h"

int initsem(key_t semkey)
{
    int status = 0, semid;
    if((semid = semget(semkey, 1, SEMPERM|IPC_CREAT|IPC_EXCL)) == -1)
    {
        if(errno == EEXIST)
            semid = semget(semkey, 1, 0);
    }
    else /* jeśli utworzony ... */
    {
        semun arg;
        arg.val = 1;
        status = semctl(semid, 0, SETVAL, arg);
    }

    if(semid == -1 || status == -1)
    {
        perror("initsem failed");
        return (-1);
    }
    /* wszystko w porządku */
    return (semid);
}
```

Włączany plik `pv.h` ma następującą zawartość:

```
/* plik nagłówkowy przykładowego semafora */

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

```
#include <sys/sem.h>
#include <errno.h>

#define SEMPERM 0600
#define TRUE 1
#define FALSE 0

typedef union _semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} semun;
```

Będziemy używać initsem w przykładach zawartych w następnych podrozdziałach.

Operacje semafora: funkcja semop

Funkcja semop wykonuje podstawowe operacje na semaforach.

Użycie

```
#include <sys/sem.h>
int semop(int semid, struct sembuf *op_array, size_t num_ops);
```

Parametr semid jest identyfikatorem zestawu semaforów, który prawdopodobnie został uzyskany za pomocą uprzedniego wywołania funkcji semget. Parametr op_array to tablica struktur sembuf, zdefiniowanych w <sys/sem.h>. Parametr num_ops jest liczbą struktur sembuf w tablicy. Każda struktura sembuf zawiera specyfikację operacji do wykonania na semaforze.

I znów, ukierunkowana na działania na zestawach semaforów funkcja semop pozwala na niepodzielne wykonywanie grupy operacji. Oznacza to, że jeśli jedna z operacji nie może być wykonana, wtedy żadna nie będzie wykonana. Jeśli nie określono inaczej, proces powinien się zablokować do czasu, dopóki nie może wykonać wszystkich operacji naraz.

Przeanalizujmy dokładniej strukturę sembuf. Zawiera ona następujące składowe:

```
unsigned short sem_num;
short sem_op;
short sem_flg;
```

Skladowa sem_num zawiera indeks semafora w zestawie. Na przykład jeśli zestaw zawiera tylko jeden semafor, wtedy sem_num musi być równa zero. Skladowa sem_op zawiera liczbę typu *signed integer*, która określa, jakie zadanie ma wykonać w rzeczywistości funkcja semop. Mogą zasimnieć trzy przypadki:

Przypadek 1: ujemna wartość sem_op

Jest to ogólna forma polecenia semafora p(), którą omawialiśmy wcześniej. Możemy w pseudokodzie następujący podsumować działanie semop (zauważ, że do reprezentacji wartości bezwzględnej zmiennej używamy ABS):

```
if( semval >= ABS(sem_op) )
{
    ustaw semval na semval - ABS(sem_op)
}
else
{
    if( (sem_flg & IPC_NOWAIT) )
        natychmiast zwróć -1
    else
    {
        czekaj, aż semval osiągnie lub przekroczy ABS(sem_op)
        następnie odejmij ABS(sem_op), jak powyżej
    }
}
```

Podstawowa idea jest taka, że funkcja semop najpierw testuje wartość semval, związaną z semaforem sem_num. Jeśli wartość semval jest wystarczająco duża, zostaje natychmiast zmniejszona. Jeśli nie, proces czeka, aż wartość semval stanie się wystarczająco duża. Jeśli jednak w sem_flg ustawiono znacznik IPC_NOWAIT, sem_op zwraca natychmiast -1 i umieszcza w errno wartość EAGAIN.

Przypadek 2: dodatnia wartość sem_op

Jest to zgodne z tradycyjną operacją v(). Wartość sem_op zostaje po prostu dodana do odpowiadającej wartości semval. Będzie obudzony inny proces czekający na nową wartość semafora.

Przypadek 3: zerowa wartość sem_op

W przypadku tej wartości sem_op funkcja będzie czekać, aż wartość semafora stanie się zerem, ale wartość semval nie jest zmieniana. Jeśli w sem_flg ustawiony zostanie znacznik IPC_NOWAIT, a wartość semval już nie jest zerowa, wtedy funkcja semop zwraca natychmiast błąd.

Znacznik SEM_UNDO

Jest to inny znacznik, który może być ustawiony w składowej sem_flg struktury sembuf. Mówi on systemowi, aby automatycznie wykonał operację cofnij (ang. *undo*), gdy proces się zakończy. Aby móc śledzić szereg takich operacji, system utrzymuje dla semafora wartość całkowitą o nazwie semadj. Ważne jest zrozumienie, że wartości semadj są alokowane na bazie procesu, więc różne procesy mają różne wartości semadj dla tego samego semafora. Kiedy stosowana jest operacja semop i ustawiony zostaje znacznik SEM_UNDO, wartość sem_num jest po prostu odejmowana od wartości semadj. Ważny jest tu znak sem_num; wartość semadj zmniejsza się, kiedy wartość sem_num jest dodatnia i powiększa się, kiedy wartość sem_num jest ujemna. Kiedy proces wychodzi, system dodaje wszystkie jego wartości semadj do odpowiedniego semafora i w ten sposób niweczy efekt wszystkich wywołań funkcji semop. Zasadniczo znacznik SEM_UNDO powinien być używany, jeśli wartości ustawiane przez proces nie mogą zachować znaczenia poza granicami istnienia tego procesu.

Przykład semafora

Będziemy teraz kontynuować przykład, który rozpoczęliśmy procedurą `initsem`. Skupia się on na dwóch procedurach: `p()` i `v()`, które są implementacją tradycyjnych operacji semafora. Najpierw przyjrzyjmy się procedurze `p()`:

```
/* p.c -- operacja p semafora */

#include "pv.h"

int p(int semid)
{
    struct sembuf p_buf;

    p_buf.sem_num = 0;
    p_buf.sem_op = -1;
    p_buf.sem_flg = SEM_UNDO;

    if(semop(semid, &p_buf, 1) == -1)
    {
        perror("p(semid) failed");
        exit(1);
    }
    return (0);
}
```

Zwróć uwagę, że używamy znacznika `SEM_UNDO`. Kod dla procedury `v()` wygląda następująco:

```
/* v.c -- operacja v semafora */

#include "pv.h"

int v(int semid)
{
    struct sembuf v_buf;

    v_buf.sem_num = 0;
    v_buf.sem_op = 1;
    v_buf.sem_flg = SEM_UNDO;

    if(semop(semid, &v_buf, 1) == -1)
    {
        perror("v(semid) failed");
        exit(1);
    }
    return (0);
}
```

Możemy zademonstrować wykorzystanie tych względnie prostych procedur do wykonania wzajemnego wykluczania się. Rozważmy następujący program:

```
/* testsem -- test procedur semafora */

#include "pv.h"
```

```
void handlesem(key_t skey);

main()
{
    key_t semkey = 0x200;
    int i;

    for(i = 0; i < 3; i++)
    {
        if(fork() == 0)
            handlesem(semkey);
    }
}

void handlesem(key_t skey)
{
    int semid;
    pid_t pid = getpid();

    if((semid = initsem(skey)) < 0)
        exit(1);

    printf("\nprocess %d before critical section\n", pid);

    p(semid);

    printf("process %d in critical section\n", pid);

    /* w rzeczywistości wykonaj coś użytecznego */
    sleep(10);

    printf("process %d leaving critical section\n", pid);

    v(semid);

    printf("process %d exiting\n", pid);
    exit(0);
}
```

Program `testsem` tworzy trzy procesy potomne, które używają `p()` i `v()` do zatrzymywania pozostałych, jeśli jeden z nich wykonuje w tym czasie krytyczną sekcję programu. Uruchomienie programu `testsem` na jednym z komputerów dalo następujące wyniki:

```
process 799 before critical section
process 799 in critical section
process 800 before critical section
process 801 before critical section
process 799 leaving critical section
process 801 in critical section
process 799 exiting
process 801 leaving critical section
process 801 exiting
process 800 in critical section
process 800 leaving critical section
process 800 exiting
```

8.3.4 Wspólna pamięć

Operacje wspólnej pamięci pozwalają dwóm albo więcej procesom korzystać wspólnie z segmentem fizycznej pamięci (oczywiście zwykle obszary danych każdych dwóch programów są całkowicie rozłączne). Stanowią one najbardziej efektywny ze wszystkich mechanizmów IPC.

Segment wspólnej pamięci musi być najpierw utworzony za pomocą funkcji systemowej `shmget`. Jeśli pamięć już istnieje, proces może przyłączyć się do niej za pomocą `shmat` i używać do własnych, tajemniczych celów. Gdy pamięć nie jest już potrzebna, proces może się odłączyć za pomocą `shmctl`.

Funkcja systemowa `shmget`

Użycie

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int permflags);
```

Wspólne segmenty pamięci są tworzone za pomocą funkcji `shmget`.

Ta funkcja odpowiada dokładnie funkcjom `msgget` i `semget`. Najbardziej interesującym parametrem jest `size`, który podaje wymaganą minimalną wielkość (w bajtach) segmentu pamięci. Parametr `key` to wartość klucza, która identyfikuje segment. Parametr `permflags` podaje prawa dostępu dla segmentu pamięci i, podobnie jak w przypadku funkcji `msgget` i `semget`, może być sumowany bitowo (OR-owany) z `IPC_CREAT` i `IPC_EXCL`.

Operacje wspólnej pamięci: funkcje `shmat` i `shmctl`

Segment pamięci utworzony przez funkcję `shmget` jest częścią pamięci fizycznej, a nie przestrzenią danych logicznych procesu. Aby z niego korzystać, proces (i każdy proces współpracujący) musi jawnie przyłączyć segment pamięci do swojej przestrzeni danych logicznych, używając funkcji `shmat`:

Użycie

```
#include <sys/shm.h>
void *shmat(int shmid, const void *daddr, int shmflags);
```

Funkcja `shmat` dołącza segment pamięci identyfikowany przez parametr `shmid` (który pochodzi z wywołania funkcji `shmget`) z ważnym adresem wywołującego procesu. Wartością zwracaną przez `shmat` jest adres (w C takie dane adresowe są zwykle przedstawiane przez wskaźnik do `void`).

Parametr `daddr` daje programistie pewną kontrolę nad wyborem adresu przez wywołanie funkcji. Jeśli jest równy `NULL`, dołączony zostaje segment z pierwszego

dostępnego adresu wybranego przez system. To oczywiście najprostszy przypadek dla programu. Jeśli parametr `daddr` nie jest równy `NULL`, dołączony będzie segment o adresie (lub blisko niego) zawartym w tym parametrze, przy czym dokładne działanie zależy od znaczników w argumencie `shmflags`. Jest to znacznie trudniejsze, ponieważ trzeba znać rozmieszczenie swojego programu w pamięci.

Parametr `shmflag` został zbudowany z dwóch znaczników: `SHM_RDONLY` i `SHM_RND`, zdefiniowanych w pliku nagłówkowym `<sys/shm.h>`. Znacznik `SHM_RDONLY` просi, aby segment był przyłączony tylko do odczytu. Znacznik `SHM_RND` – gdy jest dostępny – wpływa na sposób, w jaki `shmat` traktuje niezerową wartość parametru `daddr`. Jeśli jest on ustawiony, wywołanie zaokrągli `daddr` do granicy strony w pamięci. Jeśli nie, funkcja `shmat` będzie używać dokładnej wartości `daddr`.

Jeśli wystąpi błąd, `shmat` zwróci raczej niemilą wartość:

```
(void *) -1
```

Istnieje jeszcze jedna operacja: `shmctl`. Jest ona odwrotnością `shmat` i odłącza przydzielony segment pamięci od przestrzeni adresów logicznych procesu (to znaczy, że proces już go nie może używać)! Jej wywołanie jest proste:

```
retval = shmctl(memptr);
```

Wartość `retval` jest liczbą całkowitą i wynosi 0 w przypadku powodzenia, a -1 przy błędzie.

Funkcja systemowa `shmctl`

Użycie

```
#include <sys/shm.h>
int shmctl(int shmid, int command, struct shmid_ds *shm_stat);
```

Funkcja ta odpowiada dokładnie `msgctl`, a parametr `command` może przybierać wśród innych opcji wartości `IPC_STAT`, `IPC_SET` i `IPC_RMID`. Będziemy używać tej funkcji, z parametrem `command` ustawionym na `IPC_RMID`, w następującym przykładzie.

Przykład wspólnej pamięci: program `shmcopy`

W tym podrozdziale zbudujemy prosty przykładowy program `shmcopy`, aby zademonstrować praktyczne wykorzystanie wspólnej pamięci. Program `shmcopy` kopiuje tylko swoje standardowe wejście do swojego standardowego wyjścia, ale omija właściwość Uniksa, dzięki której wszystkie wywołania funkcji `read` i `write` blokują się wzajemnie, dopóki nie będą zakończone. Każde wywołanie `shmcopy` kończy się dwoma procesami, odczytującym i zapisującym, które mają wspólne dwa bufora zaimplementowane jako wspólny segment pamięci. Podczas gdy proces czytający odczytuje dane do pierwszego bufora, proces piszący zapisuje zawartość drugiego bufora, i odwrotnie. Ponieważ odczyty i zapisy są wykonywane równocześnie, przepustowość danych powinna być większa. To podejście

jest stosowane na przykład w programach, które sterują szybkimi strimerami taśmowymi.

Aby koordynować oba procesy i zabezpieczyć proces zapisujący przed zapisem z bufora, zanim zapelni go proces odczytujący, użyjemy dwóch semaforów dwójkowych. Niemal wszystkie programy korzystające ze wspólnej pamięci muszą używać w tej czy innej formie semaforów w celu synchronizacji (urządzenia wspólnej pamięci nie posiadają swoich własności synchronizacyjnych).

Program `shmcopy` używa następującego pliku nagłówkowego `share_ex.h`:

```
/* plik nagłówkowy dla przykładu wspólnej pamięci */

#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define SHMKEY1      (key_t)0x10 /* klucz wspólnej pamięci */
#define SHMKEY2      (key_t)0x15 /* klucz wspólnej pamięci */
#define SEMKEY       (key_t)0x20 /* klucz semafora */

/* wielkość bufora dla odczytów i zapisów */
#define SIZ 5*BUFSIZ

/* zawiera dane i licznik odczytów */
struct databuf {
    int d_nread;
    char d_buf[SIZ];
};

typedef union _semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} semun;
```

Pamiętaj, że stała `BUFSIZ` jest zdefiniowana w `<stdio.h>` i podaje współczynnik blokowania dysku systemowego. Wzorzec `databuf` pokazuje strukturę, którą będziemy umieszczać w każdym wspólnym segmencie pamięci. W szczególności składowa `d_nread` umożliwia procesowi czytającemu przekazanie poprzez segmenty pamięci liczby odczytyanych znaków do procesu piszącego.

Następny plik zawiera procedurę inicjacji dwóch wspólnych segmentów pamięci i zestawu semaforów. Zawiera on też procedurę `remobj`, która usuwa różne obiekty IPC pod koniec działania programu. Szczególnie zwróć uwagę na sposób wywołania funkcji `shmat` w celu przyłączenia segmentów wspólnej pamięci do przestrzeni adresowej procesu.

```
/* procedury inicjacji */

#include "share_ex.h"
#define IFLAGS      (IPC_CREAT | IPC_EXCL)
#define ERR        ((struct databuf *)-1)
```

```
static int shmid1, shmid2, semid;

void getseg(struct databuf **p1, struct databuf **p2)
{
    /* utwórz segment wspólnej pamięci */
    if((shmid1 = shmget(SHMKEY1, sizeof(struct databuf),
                        0600 | IFLAGS)) == -1)
        fatal("shmget");
    if((shmid2 = shmget(SHMKEY2, sizeof(struct databuf),
                        0600 | IFLAGS)) == -1)
        fatal("shmget");

    /* przyłącz segmenty wspólnej pamięci */
    if((*p1 = (struct databuf *)shmat(shmid1, 0, 0)) == ERR)
        fatal("shmat");
    if((*p2 = (struct databuf *)shmat(shmid2, 0, 0)) == ERR)
        fatal("shmat");
}

int getsem(void)           /* pobierz zestaw semaforów */
{
    semun x;
    x.val = 0;

    /* utwórz zestaw dwóch semaforów */
    if((semid = semget(SEMKEY, 2, 0600 | IFLAGS)) == -1)
        fatal("semget");

    /* ustaw wartości początkowe */
    if(semctl(semid, 0, SETVAL, x) == -1)
        fatal("semctl");
    if(semctl(semid, 1, SETVAL, x) == -1)
        fatal("semctl");
    return(semid);
}

/* usuń identyfikatory wspólnej pamięci i zestawu semaforów */
void remobj(void)
{
    if(shmctl(shmid1, IPC_RMID, NULL) == -1)
        fatal("shmctl");
    if(shmctl(shmid2, IPC_RMID, NULL) == -1)
        fatal("shmctl");
    if(semctl(semid, IPC_RMID, NULL) == -1)
        fatal("semctl");
}
```

Błędy w tych procedurach są obsługiwane za pomocą funkcji `fatal`, której używaliśmy w poprzednich przykładach. Wywołuje ona po prostu funkcję `perror`, a następnie `exit`.

Poniżej podajemy funkcję `main` dla `shmcopy`. Jest ona bardzo prosta i składa się tylko z wywołania procedury inicjacji, a następnie tworzenia procesów odczytu (proces rodzicielski) i zapisu (proces potomny). Zauważ, że właśnie proces zapisu wywołuje funkcję `remobj`, kiedy program się kończy.

```

/* shmcopy -- funkcja main */
#include "share_ex.h"

main()
{
    int semid;
    pid_t pid;
    struct databuf *buf1, *buf2;

    /* inicjuj zestaw semaforów */
    semid = getsem();

    /* utwórz i przyłącz segmenty wspólnej pamięci */
    getseg(&buf1, &buf2);

    switch(pid = fork()){
        case -1:
            fatal("fork");
        case 0: /* potomek */
            writer(semid, buf1, buf2);
            remobj();
            break;
        default: /* rodzic */
            reader(semid, buf1, buf2);
            break;
    }

    exit(0);
}

```

Funkcja main tworzy obiekty IPC przed wywołaniem fork. Zauważ, że adresy, które identyfikują wspólne segmenty pamięci (trzymane w buf1 i buf2), są znaczące dla obu procesów.

Pierwszą z interesujących procedur jest reader, która pobiera swoje wejście ze standardowego wejścia, to znaczy deskryptora pliku 0. W parametrze semid przekazywany jest do niej identyfikator zestawu semaforów, a w parametrach buf1 i buf2 – adresy dwóch segmentów wspólnej pamięci.

```

/* reader -- obsługa odczytu pliku */

#include "share_ex.h"

/* zdefiniuj p() and v() dla dwóch semaforów */
struct sembuf p1 = {0,-1,0}, p2 = {1,-1,0};
struct sembuf v1 = {0,1,0}, v2 = {1,1,0};

void reader(int semid, struct databuf *buf1,
           struct databuf *buf2)
{
    for(;;)
    {
        /* odczytaj do bufora buf1 */
        buf1->d_nread = read(0, buf1->d_buf, SZ);
        /* miejsce synchronizacji */

```

```

        semop(semid, &p1, 1);
        semop(semid, &p2, 1);

        /* testuj dla uniknięcia wstrzymania procesu piszącego */
        if(buf1->d_nread <=0)
            return;

        buf2->d_nread = read(0, buf2->d_buf, SZ);

        semop(semid, &v1, 1);
        semop(semid, &v2, 1);

        if(buf2->d_nread <=0)
            return;
    }
}

```

Struktura sembuf definiuje tu tylko operacje p() i v() dla zestawu dwóch semaforów. Jednak tym razem nie są one używane do blokowania sekcji krytycznej kodu, lecz do synchronizacji procesów odczytu i zapisu. Procedura reader używa v2 do sygnalizacji, że odczyt został zakończony i czeka, za pomocą wywołania semop z parametrem p1, aż procedura writer da sygnał, że zapis został zakończony. Będzie to bardziej jasne, gdy opiszymy procedurę writer. Możliwe są inne sposoby, wymagające czterech semaforów dwójkowych albo semaforów, które mogą przybierać więcej niż dwie wartości.

Końcową procedurą wywoływaną przez program shmcopy jest writer:

```

/* writer -- obsługa zapisu */

#include "share_ex.h"

extern struct sembuf p1, p2; /* zdefiniowany w reader.c */
extern struct sembuf v1, v2; /* zdefiniowany w reader.c */

void writer(int semid, struct databuf *buf1,
           struct databuf *buf2)
{
    for(;;)
    {
        semop(semid, &p1, 1);
        semop(semid, &v2, 1);

        if(buf1->d_nread <= 0)
            return;

        write(1, buf1->d_buf, buf1->d_nread);

        semop(semid, &p1, 1);
        semop(semid, &v2, 1);

        if(buf2->d_nread <= 0)
            return;

        write(1, buf2->d_buf, buf2->d_nread);
    }
}

```

Ponownie zwróć uwagę na użycie zestawu semaforów do koordynacji procedur reader i writer. Tym razem writer używa v2 do sygnalizacji, a czeka na p1. Ważne jest też zwrócenie uwagi, że wartości dla buf1 >d_nread i buf2 >d_nread są ustawiane przez proces czytania.

Po skompilowaniu, program shmcopy może być używany za pomocą następującego polecenia:

```
$ shmcop < big > /tmp/big
```

Ćwiczenie 8.6 Ulepsz w programie shmcopy obsługę błędów i raporty o nich (szczególnie dla wywołań read i write). Program shmcopy powinien akceptować jako argument nazwy plików w takim stylu, jak przyjmuje je program cat. Podaj konsekwencje przerwania shmcopy. Czy możesz to poprawić?

Ćwiczenie 8.7 Opracuj system przekazywania komunikatów, który używa wspólnej pamięci. Przetestuj go i porównaj wyniki z procedurami przekazywania komunikatów IPC.

8.3.5 Polecenia ipcs i ipcrm

Istnieją dwa polecenia na poziomie powłoki, które umożliwiają użycie urządzeń IPC. Pierwszym z nich jest ipcs, drukujące informację o bieżącym stanie urządzeń IPC. Oto prosty przykład:

```
$ ipcs
```

IPC status from /dev/kmem as of Wed Feb 26 18:31:31 1998

T	ID	KEY	MODE	OWNER	GROUP
Message Queues:					
Shared Memory:					
Semaphores:					
s	10	0x00000200	--ra-----	keith	users

Inne polecenie to ipcrm, używane do usuwania urządzeń IPC z systemu (pod warunkiem, że użytkownik jest właścicielem urządzenia albo super-użytkownikiem). Na przykład polecenie:

```
$ ipcrm -s 0
```

usuwa semafor związany z identyfikatorem 0, a polecenie:

```
$ ipcrm -S 200
```

usuwa semafor z wartością klucza 200.

Więcej szczegółów dostępnych opcji znajdziesz w podręczniku swojego systemu.

ROZDZIAŁ 9

Terminal

- 9.1 Wprowadzenie
- 9.2 Terminal uniksowy
- 9.3 Spojrzenie ze strony programu
- 9.4 Pseudoterminale
- 9.5 Przykład obsługi terminala: program tscript

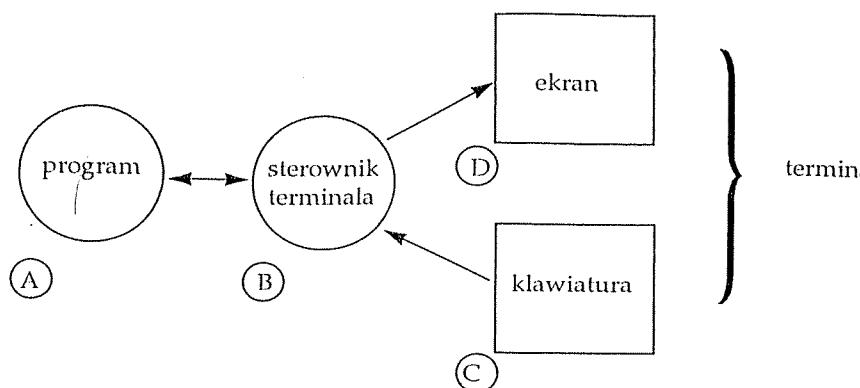
9.1 Wprowadzenie

Z każdym razem, gdy program i użytkownik oddziałują na siebie wzajemnie za pomocą terminala, dzieje się znacznie więcej, niż może wyglądać na pierwszy rzut oka. Na przykład jeśli program zapisuje ciąg znaków do urządzenia terminala, ten ciąg znaków (napis) jest najpierw przetwarzany przez część jądra, którą nazywamy sterownikiem terminala (ang. *terminal driver*). Zależnie od wartości pewnych znaczników stanu w systemie, napis może być po prostu przekazany dalej bez zmian lub zmieniony w pewien sposób przez sterownika. Zawsze dokonywana jest zamiana znaku line-feed (wysunięcie wiersza) lub newline (znak nowego wiersza) na sekwencję składającą się z dwóch znaków carriage-return, newline (powrót karetki, znak nowego wiersza). Dzięki temu każda linia zawsze zaczyna się po lewej stronie ekranu terminala lub otwartego okna.

W normalnych warunkach sterownik terminala pozwala użytkownikowi edytować pomyłki w linii wejściowej za pomocą znaków erase (kasuj) i kill (usuń). Znak erase kasuje ostatnio wpisany znak, podczas gdy znak kill usuwa wszystkie znaki aż do początku linii. Dopiero gdy użytkownik jest zadzwolony z zawartością linii i naciśnie klawisz [Return], sterownik terminala przekazuje ją do programu.

To jeszcze nie koniec historii. Na przykład, gdy napis wyjściowy dotrze już do terminala, sprzęt terminala może wyświetlić go bezpośrednio lub interpretować jako sekwencję unikową (ang. *escape-sequence*) do sterowania ekranem. Dlatego ostatnim wynikiem może być wyświetlenie komunikatu lub wyczyszczenie zawartości ekranu.

Rysunek 9.1 pokazuje wyraźniej różne składniki połączenia pomiędzy komputerem a terminaliem.



Rysunek 9.1 Połączenie pomiędzy procesem Unixa a terminaliem

Połączenie to zawiera cztery elementy:

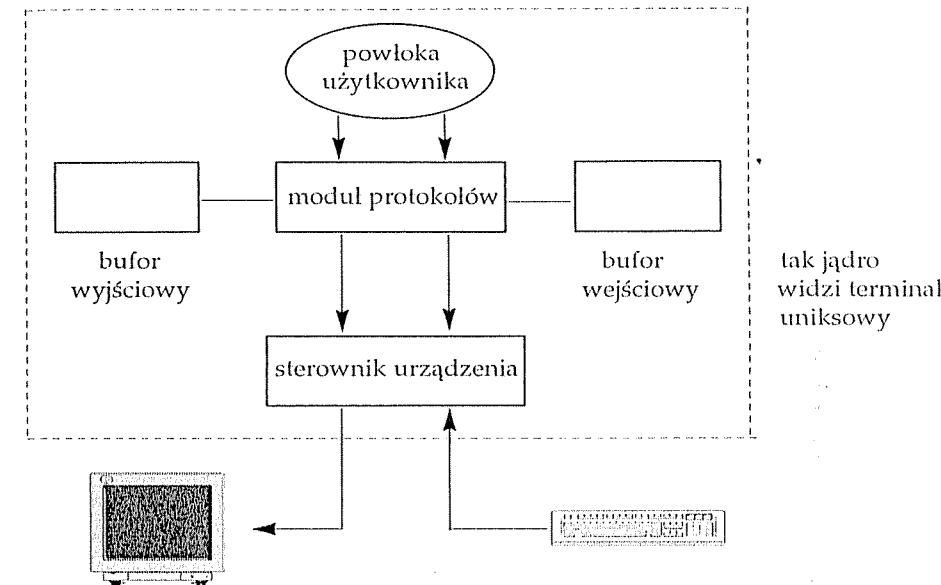
- *Program (A)* Generuje sekwencję znaków wyjściowych i interpretuje sekwencje znaków wejściowych. Może oddziaływać wzajemnie z terminaliem za pomocą takich funkcji systemowych, jak `read` lub `write`, bibliotek wysokiego poziomu jak Standardowa Biblioteka I/O lub specjalnego pakietu bibliotecznego przeznaczonego do sterowania ekranem. Oczywiście ostatecznie wszystkie operacje I/O będą wykonywane za pomocą funkcji `read` i `write`, ponieważ biblioteki wyższego poziomu muszą koniec końców wywoływać te funkcje pierwotne.
- *Sterownik terminala (B)* Główna funkcja sterownika terminala przeznaczona do transferu danych z programu do urządzenia peryferyjnego i odwrotnie. W samym jądrze Unixa terminal najczęściej składa się z dwóch głównych składników programowych: **sterownika urządzenia** (ang. *device driver*) i **modułu protokołów** (ang. *line discipline*).

Sterownik urządzenia to niskopoziomowa część oprogramowania stanowiąca interfejs do określonego sprzętu, która pozwala komputerowi komunikować się z terminaliem. Istotnie systemy najczęściej potrzebują sterowników urządzeń, aby współdziałać z więcej niż jednym typem sprzętu. Wykorzystywana tu warstwa niskiego poziomu stanowi urządzenie zapewniające, że podstawowe cechy sterownika urządzenia stają się bardziej ogólne i niezależne od sprzętu.

Oprócz tej podstawowej funkcji transmisyjnej, sterownik terminala powinien wykonywać w pewnym stopniu przetwarzanie logiczne danych wejściowych i wyjściowych, odwzorowując jedną sekwencję znaków na inną. Przetwarzanie to jest obsługiwane przez moduł protokołów. Może on także dostarczać wielu funkcji wspomagających końcowego użytkownika, takich jak edycja linii wejściowej. Rodzaj przetwarzania i wykonywanego odwzorowywania zależy od znaczników stanu przechowywanych przez moduł protokołów dla każdego portu terminala. Znaczniki te mogą być ustawiane za

pomocą grupy funkcji systemowych, które omówimy w dalszych podrozdziałach.

- *Klawiatura i ekran (C i D)* Te dwa elementy reprezentują sam terminal i uwijają jego dwoistą naturę. Węzeł (C) oznacza klawiaturę terminala i działa jako źródło danych wejściowych. Węzeł (D) oznacza ekran terminala i działa jako ujście danych wyjściowych. Jednak program może uzyskać dostęp do terminala zarówno jako źródła danych wejściowych, jak też i ujścia danych wyjściowych za pomocą tylko jednej nazwy terminala i ostatecznie pojedynczego deskryptora pliku. Aby było to możliwe, moduł protokołów utrzymuje wejściową i wyjściową kolejkę znaków dla każdego terminala. Układ ten jest pokazany na rysunku 9.2.



Rysunek 9.2 Implementacja terminala

Dotychczas zakładaliśmy, że urządzenie peryferyjne przyłączone do linii terminala to standardowy monitor ekranowy (ang. *video display unit, VDU*), ale urządzeniem peryferyjnym równie dobrze może być drukarka szeregową, ploter, punkt wejściowy sieci lub nawet inny komputer. Niemniej jednak, bez względu na naturę urządzenia peryferyjnego, może ono działać odpowiednio jako źródło wejściowego i ujście wyjściowego strumienia znaków.

W tym rozdziale skoncentrujemy się na węzłach (A) i (B) rysunku. Innymi słowy, zbadamy interakcję między programem a sterownikiem terminala na poziomie funkcji systemowych. Nie będziemy rozważać oddzielnie zagadnienia obsługi ekranu, ponieważ sterownik terminala nie odgrywa roli w budowie odpowiednich sekwencji unikowych do sterowania zawartością ekranu.

Dodamy tu jeszcze dwie uwagi. Po pierwsze, będziemy rozważać tylko normalne środowisko terminala, a nie złożoność środowiska okienkowego X-Windows lub MS-Windows. Środowiska te mają własne specyficzne problemy, którymi nie będziemy się zajmować. Po drugie, obsługa terminala pod Unixem od dawna znana jest z niekonsekwencji. Jednak XSI dostarcza standardowy zestaw interfejsu funkcji systemowych, na którym się tu skoncentrujemy.

9.2 Terminal uniksowy

Jak można było oczekwać na podstawie uwag w rozdziale 4, terminale są identyfikowane przez pliki urządzeń (i – z powodu swojej natury – traktowane jako urządzenia znakowe). W konsekwencji terminale lub ściślej porty terminali, najczęściej są dostępne za pomocą nazw plików w katalogu dev. Oto przykład typowych nazw terminali:

```
/dev/console  
/dev/tty01  
/dev/tty02  
/dev/tty03  
...
```

tty jest synonimem terminala, często używanym w Uniksie.

Z powodu uogólnienia pojęcia pliku uniksowego można uzyskać dostęp do terminali za pomocą standardowych systemowych funkcji pierwotnych jak `read` i `write`. Prawa dostępu do pliku zachowują swoje zwykłe znaczenie i kontrolują dostęp do terminali w systemie. Aby jednak te ustalenia działały rozsądnie, system zmienia prawa własności terminala, gdy użytkownik się rejestruje i wszyscy użytkownicy posiadający terminal pracują dalej.

Proces zwykle nie potrzebuje jawnie otwierać pliku terminala, aby współdziałać z użytkownikiem. Dzieje się tak, ponieważ standardowe wejście i standardowe wyjście procesu, jeśli nie zostały przekierowane, zostają przyłączone do terminala użytkownika. A więc, jeżeli standardowe wyjście nie będzie przypisane do pliku, następujący fragment kodu spowoduje wypisanie danych na ekranie terminala:

```
#define FD_STDOUT 1  
.  
. .  
.  
write(FD_STDOUT, mybuffer, somesize);
```

W tradycyjnym środowisku Uniksa terminale rejestracyjne są faktycznie otwierane jako pierwsze podczas uruchamiania systemu, pod kontrolą programu nadzorującego procesy `init`. Deskryptory plików terminala są przekazywane w dół do potomków `init` i ostatecznie każdy proces powłoki użytkownika dziedziczy trzy deskryptory plików połączone z terminaliem użytkownika. Oczywiście są to: standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach powłoki. Deskryptory te są z kolei przekazywane do każdego programu rozpoczęzanego w powloce.

9.2.1 Terminale sterujące

W normalnych okolicznościach terminal powiązany z procesem przez swoje deskryptory plików standardowych jest **terminaliem sterującym** (ang. *control terminal*) dla tego procesu i jego sesji. Terminal sterujący jest ważnym atrybutem procesu, który określa obsługę przerwań generowanych przez klawiaturę. Na przykład jeśli użytkownik naciśnie bieżący klawisz przerwania, wszystkie procesy, które rozpoznają ten terminal jako swój terminal sterujący, otrzymają sygnał `SIGINT`. Terminal sterujące, podobnie jak inne atrybuty procesu, dziedziczone są za pomocą wywołania funkcji `fork`. (Ścisłe mówiąc, terminal staje się terminaliem sterującym dla sesji, gdy otwiera go początek sesji. Dzieje się tak, jeśli terminal nie jest już powiązany z sesją i początek sesji nie nabył terminala sterującego. Wynika z tego, że proces może przełamać powiązanie ze swoim terminaliem sterującym przez zmianę swojej sesji za pomocą `setsid`. Widzieliśmy to, raczej przedwcześnie, w rozdziale 5. Teraz powinieneś zobaczyć, jak `init` wszystko organizuje.)

W przypadku, kiedy proces musi uzyskać dostęp do swojego terminala sterującego niezależnie od stanu swoich standardowych deskryptorów pliku, może być użyta nazwa pliku:

```
/dev/tty
```

Jest ona zawsze interpretowana jako terminal sterujący bieżącego procesu. W konsekwencji identyfikowany przez ten plik rzeczywisty terminal zmienia się od procesu do procesu.

9.2.2 Transmisja danych

Zasadniczym zadaniem sterownika terminala jest transmitowanie znaków pomiędzy procesem a urządzeniem terminala. W rzeczywistości jest to skomplikowane, ponieważ użytkownik może wpisać znaki w dowolnej chwili, nawet podczas operacji wyjściowej. Aby zrozumieć to położenie, powróćmy do rysunku 9.1 i wyobraźmy sobie równoczesne przekazywanie danych wzduż ścieżek od (C) do (B) i od (B) do (D). Pamiętaj, że program przedstawiony na rysunku przy węźle (A) może w danym czasie wydać pojedyncze żądanie `read` lub `write`.

Aby zarządzać równocześnie dwoma strumieniami znaków, podczas gdy tylko jeden z nich jest przetwarzany przez program użytkownika, moduł protokołów zapamiętuje dane wejściowe i wyjściowe wewnętrznych buforach. Dane wejściowe są przekazywane do programu użytkownika, kiedy wyda on żądanie `read`. Znaki wejściowe mogą być stracone, kiedy bufory utrzymywane przez jądro zostaną całkowicie zapelnione lub liczba znaków związanych z terminaliem przekroczy określone przez system maksimum, reprezentowane przez stałą `MAX_INPUT` zdefiniowaną w `<limits.h>`. Ograniczenie to wynosi typowo 255, co jest wartością wystarczającą, aby w normalnych warunkach utrała danych była rzadkością. Jednak nie ma żadnego sposobu określenia, że dane zostały stracone; system po prostu wyrzuca niezapowiedziane dodatkowe znaki.

Sytuacja wyjściowa jest nieco prostsza. Każda skierowana do terminala operacja `write` umieszcza znaki w kolejce wyjściowej. Jeśli nawet ta kolejka się zaplni, kolej-

na operacja `write` będzie zablokowana (to znaczy wstrzymana), dopóki kolejka wyjściowa nie zmniejszy się do odpowiedniego poziomu.

9.2.3 Echo znaków i wpisywanie z wyprzedzeniem

Ponieważ terminale są używane do interakcji między ludźmi a programami komputerowymi, sterownik terminala uniksowego dostarcza wiele dodatkowych ułatwień upraszczających życie zwykłym śmiertelnikom.

Być może najbardziej podstawowym z tych dodatkowych ułatwień jest echo znaków. Przedtem wszystkim pomaga ono zobaczyć 'A' na ekranie, gdy wprowadzisz „A” z klawiatury. Terminal połączony z systemem Unix pracuje zwykle w trybie pełnego duplexu (ang. *full-duplex*), co oznacza, że za echo znaków odpowiada system Unix, a nie terminal. W konsekwencji, zaraz po wpisaniu znak jest transmitowany przez terminal do systemu Unix. Kiedy zostanie odebrany przez system, moduł protokołów natychmiast umieszcza jego kopię w kolejce wyjściowej do tego terminala. Wtedy jest on powtarzany na ekranie terminala. Oznacza to, że znak (na rysunku 9.1) jest przesyłany najpierw wzduż ścieżki od (C) do (B), a następnie odsyłany natychmiast wzduż ścieżki od (B) do (D). Wszystko to może wystąpić, zanim program (A) będzie gotowy do odczytania znaku. Prowadzi to w Unixie do interesującego zjawiska: jeśli użytkownik wpisuje coś w tym samym czasie, gdy program wyświetla komunikat na ekranie, echo znaków wejściowych pojawia się w środku komunikatu wyjściowego. W innych systemach echo znaków wejściowych może być wstrzymane, dopóki program nie jest gotowy do czytania tych znaków.

9.2.4 Tryb kanoniczny, edycja linii i znaki specjalne

Terminal może być ustawiany w różne tryby (stosownie do typu dołączonego do niego programu) obsługiwane przez prawidłowy moduł protokołów. Na przykład edytor ekranowy chce mieć maksymalną kontrolę nad ekranem i dlatego ustawia terminal w stan surowy, w którym moduł protokołów po prostu przekazuje znaki do programu w postaci, w jakiej nadchodzą, bez żadnego przetwarzania.

Jednak Unix nie byłby środowiskiem programistycznym, gdyby wszystkie programy musiały martwić się o szczegóły terminala sterującego. Tak więc moduł protokołów standardowego terminala dostarcza trybu operacji szczególnie dopasowanego do prostego zastosowania interakcyjnego, ukierunkowanego na linie. Jest on opisany jako **tryb kanoniczny** (ang. *canonical mode*) i używany przez powłokę, edytor ed i podobne programy.

W trybie kanonicznym, gdy zostaną naciśnięte pewne klawisze, sterownik terminala wykonuje specjalne działania. Wiele z tych działań związanych jest z edycją linii. W następstwie tego, kiedy terminal znajduje się w swoim stanie kanonicznym, wejście dostępne jest dla programu tylko w formie kompletnych linii. (Później powiemy o tym więcej.)

W trybie kanonicznym najbardziej znanym klawiszem edycyjnym jest znak `erase` (kasuj). Naciśnięcie go powoduje skasowanie poprzedniego znaku w linii. Na przykład wiersz polecenia:

`$ whp<erase>o`

z następującym po nim znakiem nowego wiersza spowoduje, że sterownik terminala wyśle do powłoki napis `who`. Jeśli terminal jest prawidłowo ustawiony, znak `p` powinien także zostać fizycznie usunięty z ekranu.

Znak `erase` może być ustawiony przez użytkownika na dowolną wartość ASCII. Najczęściej używany do tego celu jest znak ASCII `backspace` (znak cofania).

Najprostszy sposób zmiany tego znaku na poziomie powłoki to użycie polecenia `stty`. Na przykład:

`$ stty erase "^h"`

ustawia znak `erase` na [Ctrl+H], co jest inną nazwą znaku cofania. Zauważ, że zwykle zamiast samego znaku [Ctrl+H] możesz wpisać napis "`^h`".

Poniżej podajemy systematyczny opis innych znaków, które w trybie kanonicznym mają specjalne znaczenie dla powłoki, jak zdefiniowano w *XSI*. Jeśli nie zaznaczono inaczej, ich dokładne wartości mogą być ustawiane przez użytkownika albo administratora systemu.

`kill` Powoduje usunięcie wszystkich znaków aż do początku linii. Sekwencja wejściowa:

`$ echo<kill>who`

z następującym po niej znakiem nowego wiersza spowoduje więc wykonanie polecenia `who`. Domyślną wartością `kill` jest [Ctrl+?]. Najczęstszy alternatywami są [Ctrl+X] i [Ctrl+U]. Znak `kill` może być ponownie ustawiony za pomocą polecenia powłoki:

`$ stty kill <nowy_znak>`

`intr` Znak przerwania. Jeśli użytkownik go naciśnie, do programu czekającego z terminala i wszystkich innych procesów rozpoznających ten terminal jako swój terminal sterujący, wysyłany zostanie sygnał `SIGINT`. Program podobnie jak powłoka jest wystarczająco czulny, aby przechwycić ten sygnał, ponieważ domyślnym działaniem po jego odebraniu jest zakończenie programu. Domyślną wartością znaku `intr` jest znak ASCII `delete`, czasami nazywany także `DEL`. Najczęstsza alternatywa to [Ctrl+C]. Aby zmienić bieżącą wartość `intr` na poziomie powłoki, użyj polecenia:

`$ stty intr <nowy_znak>`

Bardziej szczegółowe omówienie obsługi sygnałów znajdziesz w rozdziale 6.

`quit` Jeśli ten znak zostanie wprowadzony przez użytkownika, spowoduje wysłanie do grupy procesów związanych z terminaliem sygnału `SIGQUIT`. Podobnie jak poprzednio, powłoka przechwytuje ten sygnał, pozostawiając programowi użytkownika podjęcie odpowiednich działań. Jak widzieliśmy w rozdziale 6, skutkiem jest zwykle zrzut rdzenia (ang. *core dump*), co powoduje zapisanie na dysku przestrzeni pamięci programu i zakończenie wykonywania programu komunikalem `Quit – core dumped` (wyjście – rdzeń)

zrzucony). Zwykle znakiem *quit* jest znak ASCII *FS* lub [Ctrl+\]. Może to być zmienione za pomocą polecenia:

```
$ stty quit <nowy_znak>
```

eof Ten znak jest używany do oznaczenia końca strumienia wejściowego z terminala (w tym celu musi być wprowadzony sam w nowej linii). Na przykład może to być znak wymuszający wyrejestrowanie się z systemu (ang. *logout*). Standardowym początkowym ustawieniem jest znak ASCII *eot*, znany także jako [Ctrl+D]. Może to być zmienione za pomocą polecenia:

```
$ stty eof <nowy_znak>
```

nl Jest to normalny ogranicznik linii. Zawsze ma wartość ASCII *line-feed* (nowy wiersz), co w C jest znakiem *newline*. Nie może on być ustawiany ani zmieniany przez użytkownika. Przy terminalach, które wysyłają znak *carriage-return* (powrót karetki) zamiast *line-feed*, moduł protokołów może być ustawiony na odwzorowywanie znaków powrota karetki na znaki nowego wiersza.

eol Jest to dodatkowy ogranicznik linii, działający jak *nl*. Zwykle nie jest on używany, a wartość domyślną stanowi znak ASCII *NULL*.

stop Najczęściej znak ten ma wartość [Ctrl+S] i w niektórych implementacjach nie może być zmieniany przez użytkownika. Jest używany do chwilowego zawieszenia wyjścia zapisującego do terminala. Szczególnie przydaje się, gdy używamy staromodnego terminala VDU, ponieważ może być wykorzystany do zawieszenia wyjścia, zanim zniknie ono na zawsze na górze ekranu terminala.

start Zwykle znak ten ma wartość [Ctrl+Q]. I znów, to czy może być zmieniony, zależy od implementacji. Jest używany do ponownego uruchomienia wyjścia, zatrzymanego uprzednio przez [Ctrl+S]. Jeśli znak [Ctrl+S] nie został wprowadzony, znak [Ctrl+Q] jest ignorowany.

susp Jeśli ten znak zostanie wpisany przez użytkownika, powoduje wysłanie sygnału *SIGTSTP* do grupy procesów powiązanych z terminaliem. Powoduje zawieszanie bieżącej grupy procesów pierwszoplanowych i umieszczenie jej w tle. Zwykle znakiem *suspend* jest [Ctrl+Z]. On także może być zmieniony za pomocą polecenia:

```
$ stty susp <nowy_znak>
```

Przy wprowadzaniu poleceń można uniknąć działania znaków *erase*, *kill* i *eof* poprzedzając je znakiem odwrotnego ukośnika (\). Kiedy to zrobimy, funkcja związana z danym znakiem nie będzie wykonywana i zostanie on przesłany do programu czytającego. Na przykład linia:

```
aal<erase>b<erase>c
```

spowoduje wysłanie *aal<erase>c* do programu, który aktualnie czyta z terminala.

9.3 Spojrzenie ze strony programu

Dotychczas omawialiśmy oferowane przez sterownik terminala ułatwienia pod względem interfejsu użytkownika. Teraz będziemy rozważać sprawę z punktu widzenia programu używającego terminala.

9.3.1 Funkcja systemowa open

Funkcja *open* może być używana do otwarcia modułu protokołów w podobny sposób, jak zwykłego pliku dyskowego. Na przykład:

```
fd = open("/dev/tty0a", O_RDWR);
```

Próba otwarcia terminala nie powróci jednak, dopóki nie będzie ustanowione połączenie. Dla terminala ze sterowaniem modelem oznacza to, że funkcja *open* nie wróci, aż sygnały sterujące modemem zostaną prawidłowo ustawione i właściwa będzie detekcja nośnej (ang. *carrier detect*), co może potrwać lub w ogóle się nie udać.

Następująca procedura używa procedury *alarm* (wprowadzonej w rozdziale 6) do wymuszenia czasu oczekiwania (ang. *timeout*), jeśli funkcja *open* nie powróci w rozsądny czasie.

```
/* ttyopen -- otwarcie z czasem oczekiwania */

#include <stdio.h>
#include <signal.h>

#define TIMEOUT 10
#define FALSE 0
#define TRUE 1

static int timeout = FALSE;
static char *termname;

static void settimeout(void)
{
    fprintf(stderr, "timeout on opening %s\n", termname);
    timeout = TRUE;
}

int ttyopen(char *filename, int flags)
{
    int fd = -1;
    static struct sigaction act, oact;

    termname = filename;

    /* ustaw znacznik czasu oczekiwania */
    timeout = FALSE;
    /* ustaw działanie SIGALRM */
    act.sa_handler = settimeout;
    sigfillset(&(act.sa_mask));
    sigaction(SIGALRM, &act, &oact);
```

```

alarm(TIMEOUT);

fd = open(filename, flags);

/* przywrócić poprzedni stan */
alarm(0);
sigaction(SIGALRM, &oact, &act);

return (timeout ? -1 : 0);
}

```

9.3.2 Funkcja systemowa read

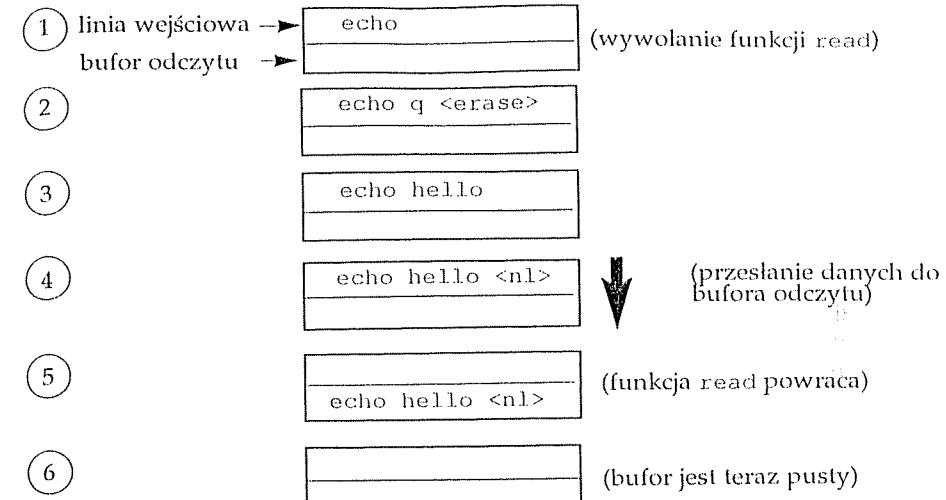
Funkcja systemowa `read` jest jedną z pierwotnych funkcji dostępu do pliku, używaną znacznie częściej w odniesieniu do pliku urządzenia terminala niż do zwykłego pliku dyskowego. Dzieje się tak szczególnie, gdy terminal znajduje się w trybie kanonicznym, przeznaczonym do zwykłego użytku interakcyjnego. W tym stanie zasadniczą jednostką wejściową staje się linia. W konsekwencji znaki nie mogą być odczytane z linii przez program, dopóki użytkownik nie naciśnie klawisza [Return], interpretowanego przez system jako znak nowego wiersza. Co równie ważne, wywołanie funkcji `read` zawsze powróci po znaku nowego wiersza, nawet jeśli liczba znaków w linii jest mniejsza niż liczba znaków żądana w wywołaniu `read`. Jeśli wpisany został tylko znak [Return] i do systemu została wysłana pusta linia, odpowiednie wywołanie `read` zwróci wartość 1, ponieważ programowi został udostępniony sam znak nowego wiersza. Dlatego do wykrycia końca pliku (to znaczy wpisania znaku `eof`) wciąż może być używany zwrot zerowej wartości.

Po raz pierwszy widzieliśmy ten rodzaj interakcji pomiędzy funkcją `read` a sterownikiem terminala w przykładzie `i.o` z rozdziału 2. Ponieważ temat zasługuje jednak na bardziej szczegółowe wyjaśnienie, rozważmy instrukcję:

```
nread = read(0, buffer, 256);
```

Jeśli standardowe wejście procesu jest pobierane ze zwykłego pliku, interpretacja tego wywołania będzie prosta: jeśli w pliku pozostało więcej niż 256 znaków, wywołanie funkcji `read` zwróci w tablicy `buffer` dokładnie 256 znaków. Ponieważ zależności między funkcją `read` a terminaliem są nieco bardziej skomplikowane, do objaśnienia użyjemy rysunku 9.3, który pokazuje w skrócie interakcję między programem a użytkownikiem.

Rysunek przedstawia możliwą sekwencję działań w sytuacji, gdy do terminala została zastosowana funkcja `read`. Na każdym etapie pokazane są dwa prostokąty: górny – pokazuje widziany przez sterownika bieżący stan linii wejściowej, natomiast dolny, nazwany buforem wejściowym, pokazuje dane aktualnie udostępnione procesowi do odczytu. Powinniśmy podkreślić, że rysunek przedstawia schemat logiczny tylko z punktu widzenia użytkownika procesu. Jednak większość implementacji sterownika terminala także używa dwóch buforów albo układu kolejek, co nie jest o wiele bardziej skomplikowane niż schemat na rysunku.



Rysunek 9.3 Etapy czytania z terminala w trybie kanonicznym

Etap 1 przedstawia sytuację w chwili wywołania funkcji `read`. W tym miejscu użytkownik wprowadził już napis `echo`, ale ponieważ nie wpisano żadnego znaku nowego wiersza, w buforze odczytu nie ma żadnych danych i wykonanie procesu zostało zawieszone.

W etapie 2 użytkownik wpisał znak `q`, a następnie zmienił zamiar i nacisnął bieżący klawisz `erase`, usuwając ten znak z linii wejściowej. Ta część rysunku uwydatnia, jak może być wykonywana edycja linii wejściowej bez udziału programu, który wykonuje odczyt.

W etapie 3 linia wejściowa jest kompletna, z wyjątkiem końcowego znaku nowego wiersza. Część rysunku oznaczona jako etap 4 pokazuje chwilę, w której został wprowadzony znak nowego wiersza i sterownik terminala przekazał linię wejściową, łącznie z końcowym znakiem nowego wiersza, do bufora odczytu. Prowadzi to do etapu 5, gdzie cała linia wejściowa stała się dostępna dla funkcji `read`. W procesie, który dokonał wywołania funkcji `read`, to wywołanie powraca, zwracając dla `nread` wartość 11. Etap 6 pokazuje położenie zaraz po zaspokojeniu tym sposobem wywołania funkcji; linia wejściowa i bufor odczytu są chwilowo puste.

Następny przykład umacnia powyższe rozważania. Skupia się on na całkowicie trywialnym programie `read_demo`, który ma jedną wyróżniającą cechę: mała wielkość bufora, używanego do pobierania danych ze standardowego wejścia.

```

/* read_demo – interakcja read ze sterownikiem terminala */

#include <sys/types.h>

#define SMALLSZ 10

```

```
main(int argc, char **argv)
{
    ssize_t nread;
    char smallbuf[SMALLSZ+1];

    while((nread = read(0, smallbuf, SMALLSZ)) > 0)
    {
        smallbuf[nread] = '\0';
        printf("nread:%d %s\n", nread, smallbuf);
    }
}
```

Jeśli ten program zostanie wykonany z następującym wejściem z klawiatury:

```
1
1234
This is a much longer line.
<EOF>
```

powstanie następujący dialog:

```
1
nread:2 1
```

```
1234
nread:5 1234
```

This is a much longer line.

```
nread:10 This is a
nread:10 much longe
nread:8 r line.
```

Zwrć uwagę, że aby wchłonąć najdłuższą linię, potrzeba kilku kolejnych odczytów. Zauważ też, że rysunek dla `nread` zawiera na końcu każdej linii znak nowego wiersza. I znów, dla większej przejrzystości, nie pokazujemy tego jawnie. Co się dzieje, jeśli terminal nie jest w trybie kanonicznym? W tym przypadku program, aby w pełni kontrolować wejście, musi ustawać dodatkowe zmienne stanu powiązane z terminalem. Wykonuje to za pomocą zestawu funkcji systemowych, które omówimy szczegółowo później.

Ćwiczenie 9.1 Wypróbuj przykład `read_demo` z tymi samymi danymi wejściowymi, ale skierowanymi z pliku.

9.3.3 Funkcja systemowa write

Dopóki interakcja z terminalem działa, funkcja `write` jest prosta. Należy tylko zaznaczyć, że ulegnie ona zablokowaniu, jeśli wyjściowa kolejka dla terminala będzie pełna. Wykonanie programu zostanie wznowione, gdy tylko liczba znaków w kolejce spadnie poniżej pewnego poziomu granicznego.

9.3.4 Funkcje systemowe `ttyname` i `isatty`

Wprowadźmy teraz dwie użyteczne funkcje narzędziowe, których będziemy używać w dalszych przykładach. Funkcja `ttyname` zwraca nazwę urządzenia terminala związanego z otwartym deskryptorem pliku terminala, podczas gdy `isatty` zwraca 1 (czyli `true` – według określenia C), jeśli deskryptor pliku opisuje urządzenie terminala, a 0 (`false`) w przypadku przeciwnym.

Użycie

```
#include <unistd.h>
char* ttyname(int filedes);
int isatty(int filedes);
```

W obu przypadkach `filedes` jest deskryptorem otwartego pliku. Jeśli `filedes` nie reprezentuje terminala, funkcja `ttyname` zwraca NULL. W przeciwnym razie wartość zwracana przez `ttyname` wskazuje na obszar danych statycznych, który jest nadpisywany przez każde wywołanie `ttyname`.

Następująca przykładowa procedura `what_tty` drukuje, jeśli jest to możliwe, nazwę terminala związanego z deskryptorem pliku:

```
/* what_tty -- drukuje nazwę terminala */
void what_tty(int fd)
{
    if(isatty(fd))
        printf("fd %d => %s\n", fd, ttyname(fd));
    else
        printf("fd %d => not a terminal!\n", fd);
```

Ćwiczenie 9.2 Zmodyfikuj procedurę `ttyopen` z ostatniego podrozdziału w ten sposób, aby zwracała deskryptor pliku tylko dla pliku specjalnego terminala, a nie dla pliku dyskowego czy pliku innego typu. Do sprawdzania wykorzystaj funkcję `isatty`. Czy istnieje inny sposób osiągnięcia tego celu?

9.3.5 Zmiana charakterystyki terminala: struktura `termios`

Na poziomie powłoki użytkownik może wywolać polecenie `stty`, aby zmienić charakterystykę modułu protokołów terminala. Program może zrobić to samo używając struktury `termios` w połączeniu z odpowiednimi funkcjami. Zwrć uwagę, że starsze systemy używają funkcji systemowej `ioct1` (jej nazwa oznacza kontrolę I/O), jak podawaliśmy w pierwszym wydaniu tej książki.¹ Funkcja `ioct1` ma zbyt ogólne zastosowanie i została teraz podzielona na bardziej specyficzne funkcje. Wszystkie one dostarczają ogólnego interfejsu programistycznego dla wszystkich portów komunikacji asynchronicznej Uniksa, niezależnie od rodzaju podstawowego sprzętu.

1. Por. przyp. ze s. VII.

Struktura `termios` może być traktowana jako reprezentacja możliwych stanów terminala, odpowiadająca znacznikom stanu pamiętanym w systemie dla każdego urządzenia terminala. Dokładną definicję tej struktury omówimy niebawem. Za pomocą wywołania funkcji `tcgetattr` struktura `termios` może być wypełniona bieżącymi ustawieniami terminala. Funkcja ta jest zdefiniowana następująco:

Użycie

```
#include <termios.h>
int tcgetattr(int ttyfd, struct termios *tsaved);
```

Ta szczególna funkcja zachowuje obecny stan terminala związanego z `ttyfd` w strukturze `termios` o nazwie `tsaved`. Parametr `ttyfd` musi być deskryptorem pliku, który opisuje terminal. Podobnie funkcja `tcsetattr`:

Użycie

```
#include <termios.h>
int tcsetattr(int ttyfd, int actions,
             const struct termios *tnew);
```

ustawia moduł protokołów reprezentowany przez `ttyfd` w nowy stan, przedstawiony przez `tnew`. Drugi parametr tej funkcji, `actions`, określa jak i kiedy powinny być ustawione nowe atrybuty terminala. Istnieją trzy możliwe działania, zdefiniowane w `<termios.h>`:

- | | |
|-----------|--|
| TCSANOW | Skutek jest natychmiastowy, co może spowodować problemy, jeśli sterownik terminala zapisuje w tym czasie do terminala i zostanie zmieniony znacznik wyjściowy w <code>tnew</code> . |
| TCSADRAIN | Wykonuje tę samą funkcję, co TCSANOW. Jednak przed ustawieniem nowych parametrów czeka na opróżnienie bieżącej kolejki wyjściowej. W konsekwencji, wartość ta powinna być używana kiedy zmieniamy parametry powiązane z wyjściem do terminala. |
| TCSAFLUSH | Jest podobne do TCSADRAIN – czeka na opróżnienie kolejki wyjściowej, a następnie opróżnia kolejkę wejściową przed ustawieniem parametrów modułu protokołów na wartości zawarte w <code>tnew</code> . |

Następujące dwie procedury wykorzystują powyższe funkcje systemowe. Procedura `tsave` zachowuje bieżące parametry związane z terminaliem sterującym procesu, a procedura `tback` przywraca ostatni zestaw zachowanych parametrów. Zmienna logiczna (boole'owska) `saved` jest używana do powstrzymania funkcji `tback` przed ustawieniem stanu końcowego, jeśli funkcja `tsave` nie była użyta.

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

```
#define SUCCESS      0
#define ERROR       (-1)

/* tsaved powinna zawierać parametry terminala */
static struct termios tsaved;

/* TRUE jeśli parametry zachowane */
static int saved = 0;

int tsave(void)
{
    if(isatty(0) && tcgetattr(0,&tsaved) >= 0)
    {
        saved = 1;
        return (SUCCESS);
    }
    return (ERROR);
}

int tback(void)           /* przywraca stan terminala */
{
    if( !isatty(0) || !saved)
        return (ERROR);

    return tcsetattr(0, TCSAFLUSH, &tsaved);
}
```

Te dwie procedury mogą być użyte do ograniczenia części kodu, który chwilowo zmienia stan terminala, jak następuje:

```
#include <stdio.h>

main()
{
    if(tsave() == -1)
    {
        fprintf(stderr, "couldn't save terminal parameters\n");
        exit(1);
    }
    /* wykonaj właściwą pracę */
    tback();
    exit(0);
}
```

Definicja struktury `termios`

Omówimy teraz szczegółowo strukturę `termios`. Wzorzec tej struktury znajduje się w pliku włączanym `<termios.h>` i zawiera następujące składowe:

```
tcflag_t   c_iflag;          /* tryby wejściowe */
tcflag_t   c_oflag;          /* tryby wyjściowe */
tcflag_t   c_cflag;          /* tryby sterujące */
tcflag_t   c_lflag;          /* tryby modułu protokołów */
cc_t      c_cc[NCCS];        /* znaki sterujące */
```

Najłatwiej zbadać tę strukturę, zaczynając od jej ostatniej składowej `cc_c`.

Tablica c_cc

Znaki edytujące linię, które omawialiśmy w podrozdziale 9.2.4, w rzeczywistości znajdują się w tablicy `c_cc`. Ich względne pozycje są określone przez stałe zdefiniowane w `<termios.h>`. Wszystkie wartości zdefiniowane przez `XSI` są pokazane w tabeli 9.1. Wielkość tablicy jest określona przez stałą `NCCS`, która też została zdefiniowana w `<termios.h>`.

Tabela 9.1 Kody znaków sterujących

stała	znaczenie
VINTR	klawisz przerwania
VQUIT	klawisz zakończenia
VERASE	znak kasowania
VKILL	znak usunięcia (kasowania linii)
VEOF	znak końca pliku
VEOL	opcjonalny znaczek końca linii
VSTART	znak rozpoczęcia
VSTOP	znak zatrzymania
VSUSP	znak zawieszenia

Następujący fragment programu pokazuje, jak zmienić wartość znaku `quit` dla terminala powiązanego ze standardowym wejściem (deskryptor pliku równy 0) :

```
struct termios tdes;
/* pobierz początkową charakterystykę terminala */
tcgetattr(0, &tdes);

tdes.c_cc[VQUIT] = '\031'; /* CTRL-Y */

/* ustaw parametry terminala */
tcsetattr(0, TCSAFLUSH, &tdes);
```

Przykład ten służy do ilustracji bezpiecznego podejścia do zmiany stanu terminala. Po pierwsze, pobierz aktualny stan terminala. Po drugie, zmień tylko te parametry, którymi jesteś zainteresowany, bez zmiany innych. Po trzecie, zmień stan terminala za pomocą zmodyfikowanej struktury `termios`. Jak widzieliśmy, warto jest także zachować oryginalną wartość w celu przywrócenia stanu terminala przed wyjściem programu, w przeciwnym przypadku późniejsze programy mogą napotkać niespodzianki.

Pole c_cflag

Pole `c_cflag` definiuje sprzętowe sterowanie terminala. Zwykle proces powinien pozostawić w spokoju pole `c_cflag` swojego terminala sterującego. Pole to jest przydatne w takich zastosowaniach, jak pakiety komunikacyjne lub gdy program otwiera dodatkową linię terminala, taką jak port drukarki. Wartości `c_cflag` są budowane za pomocą sumy bitowej (OR) stałych zdefiniowanych w `<termios.h>`.

Zasadniczo każda stała reprezentuje pojedyńczy bit w znaczniku pola, który może być ustawiony lub wyzerowany. Istnieje wiele takich stałych, których nie będziemy omawiać w pełni (szczegóły znajdziesz w podręczniku swojego systemu). Istnieją jednak cztery funkcje, które pozwalają pobrać i ustawić szybkość wejściową oraz wyjściową bez troszczenia się o stan pozostałych bitów.

Użycie

```
#include <termios.h>

/* ustawia szybkość wejściową */
int cfsetispeed(struct termios *tdes, speed_t speed);

/* ustawia szybkość wyjściową */
int cfsetospeed(struct termios *tdes, speed_t speed);

/* pobiera szybkość wejściową */
speed_t cfgetispeed(const struct termios *tdes);

/* pobiera szybkość wyjściową */
speed_t cfgetospeed(const struct termios *tdes);
```

Następujący przykład ustawia szybkość terminala na 9600 bodów. Wartość B9600 jest zdefiniowana w `<termios.h>`.

```
struct termios tdes;

/* pobierz początkową charakterystykę terminala */
tcgetattr(0, &tdes);

/* zmień szybkość wejścia i wyjścia */
cfsetispeed(&tdes, B9600);
cfsetospeed(&tdes, B9600);
```

Oczywiście nie będzie to miało żadnego skutku, dopóki nie wywołamy funkcji `tcsetattr`, jak poniżej:

```
tcsetattr(0, TCSAFLUSH, &tdes);
```

Następny przykład włącza generację i sprawdzanie parzystości przez bezpośrednie ustawienie bitów:

```
tdes.c_cflag |= (PARENB | PARODD);
tcsetattr(0, TCSAFLUSH, &tdes);
```

Występuje tu znaczek `PARENB`, który włącza kontrolę parzystości. Znaczek `PARODD` wskazuje, że wymagana jest parzystość nieparzysta (ang. *odd*). Jeśli znaczek `PARODD` jest wyłączony, a `PARENB` ustawiony, wtedy przyjmowana jest parzystość parzysta (ang. *even*). (Termin **parzystość** oznacza użycie w transmisji danych bitów kontrolnych. Możliwy jest tylko jeden taki bit kontrolny w znaku, ponieważ zestaw znaków ASCII wykorzystuje siedem z ośmiu bitów używanych przez większość komputerów do zapamiętania bajtu. Wartość bitu kontrolnego może być używana do ustawienia całkowitej liczby bitów ustawionych w bajcie na wartość nieparzystą lub parzystą. Programista może alternatywnie wybrać całkowite ignorowanie parzystości).

Pole c_iflag

Pole `c_iflag` struktury `termios` opisuje podstawowe sterowanie wejściem terminala. I znów nie będziemy omawiać wszystkich możliwych ustawień, lecz tylko te, które są używane najczęściej.

Trzy ze znaczników związanych z tym polem dotyczą sposobu traktowania powrotu karetki. Mogą być przydatne przy terminalach, które wysyłają sekwencję ze znakiem powrotu karetki oznaczającym koniec linii. (Unix oczywiście oczekuje jako końca linii znaku ASCII wysunięcia wiersza lub nowego wiersza). Wspomnianymi znacznikami są:

`INLCR` odwzoruj, czyli zamień znak nowego wiersza na powrót karetki

`IGNCR` ignoruj powrót karetki

`ICRNL` odwzoruj znak powrotu karetki na znak nowego wiersza

Trzy inne znaczniki pola `c_iflag` dotyczą sterowania transmisją:

`IXON` pozwól na sterowanie wyjściem za pomocą znaków start/stop

`IXANY` pozwól dowolnemu znakowi na wznowienie wyjścia

`IXOFF` pozwól na sterowanie wejściem za pomocą znaków start/stop

Znacznik `IXON` daje użytkownikowi możliwość sterowania wyjściem. Jeśli jest ustawiony, użytkownik może zatrzymywać wyjście za pomocą `[Ctrl+S]`. Znak `[Ctrl+Q]` spowoduje ponowne uruchomienie wyjścia. Jeśli ustawiony jest też znacznik `IXANY`, do ponownego uruchomienia wyjścia można wykorzystać dowolny znak, chociaż zasadniczo do zatrzymania wyjścia musi być użyty znak `[Ctrl+S]`. Jeśli ustawiony jest znacznik `IXOFF`, system sam będzie transmitował znak zatrzymania (jak zwykle `[Ctrl+S]`) do terminala, kiedy jego bufor wejściowy jest prawie pełny. Gdy system będzie znów gotowy do akceptacji danych, w celu ponownego uruchomienia wyjścia zostanie wysłany znak `[Ctrl+Q]`.

Pole c_oflag

Pole `c_oflag` określa traktowanie wyjścia systemu. Najważniejszym znacznikiem jest tu `OPOST`. Jeśli nie został ustawiony, znaki wyjściowe są transmitowane bez zmian. Jeśli natomiast został ustawiony, znaki są przetwarzane w sposób wskazywany przez pozostałe ustawione znaczniki tego pola. Kilka znaczników dotyczy sposobu przetwarzania znaku powrotu karetki w wyjściu do terminala:

`ONLCR` zmień znak nowego wiersza na znaki powrotu karetki i nowego wiersza

`OCRNL` zmień znak powrotu karetki na znak nowego wiersza

`ONOCR` żadnego wyjścia znaku powrotu karetki w kolumnie 0

`ONLRET` znak nowego wiersza wykonyuje funkcję znaku powrotu karetki

Jeśli znacznik `ONLCR` został ustawiony, znaki nowego wiersza są zamieniane na sekwencję znaków: powrót karetki, nowy wiersz. Dzięki temu każda linia zaczyna się po lewej stronie ekranu. I odwrotnie, jeśli ustawiony został znacznik `OCRNL`,

wtedy znaki powrotu karetki są zamieniane na znaki nowego wiersza. Znacznik `ONLRET` mówi sterownikowi terminala, że dla używanego typu terminala sam znak nowego wiersza wykonuje funkcję powrotu karetki. Jeśli ustawiony został znacznik `ONOCR`, przy wysyłaniu linii wyjściowej o zerowej długości nie jest wysyłany znak powrotu karetki.

Niemal wszystkie pozostałe znaczniki składowej `c_oflag` dotyczą opóźnień w transmisji związanych z określonymi znakami, jak znaki nowego wiersza, tabulacji, wysunięcia strony itp. Opóźnienia te uwzględniają ruchy mechaniczne albo przewinięcie zawartości ekranu, które trwają określony czas. I znów, szczególnie możesz sprawdzić w podręczniku swojego systemu.

Pole c_lflag

Prawdopodobnie najbardziej interesującą dla programistów składową struktury `termios` jest pole `c_lflag`. Jest ono używane przez bieżący moduł protokołów do sterowania funkcjami terminala. Dostępne są następujące znaczniki:

<code>ICANON</code>	wejście kanoniczne, ukierunkowane na linię
<code>ISIG</code>	włącz przetwarzanie przerwań
<code>IEXTEN</code>	włącz rozszerzone (zależne od implementacji) przetwarzanie znaków wejściowych
<code>ECHO</code>	włącz podstawowe powtarzanie wejścia
<code>ECHOE</code>	powtórz znak kasowania jako sekwencję znaków cofanie-spacja-cofanie
<code>ECHOK</code>	powtórz znak nowego wiersza po usunięciu
<code>ECHONL</code>	powtórz znak nowego wiersza
<code>NOFLSH</code>	włącz opróżnianie bufora po przerwaniu
<code>TOSTOP</code>	wyslij sygnał <code>SIGTTOU</code> dla wyjścia w tle

Jeśli znacznik `ICANON` został ustawiony, wykonywane jest przetwarzanie kanoniczne. Jak widzieliśmy wcześniej, włącza to użycie znaków edycji linii i gromadzi wejście w liniach, zanim będzie mogło być odczytane. Jeśli znacznik `ICANON` nie został ustawiony, terminal znajduje się w trybie surowym, zwykle związanym z oprogramowaniem zorientowanym na ekran i pakietami komunikacyjnymi. Wywołanie funkcji `read` będzie teraz zaspokajane bezpośrednio z kolejki wejściowej. Innymi słowy, podstawową jednostką wejściową staje się pojedynczy znak, a nie linia logiczna. Program może wybrać czytanie danych znak po znaku (przydatne przy pełnoekranowych edytorech) lub wielkimi blokami stałej wielkości (przydatne dla oprogramowania transmisyjnego). Jednak programista musi wtedy określić dwa dodatkowe parametry, aby w pełni kontrolować zachowanie funkcji `read`. Są to `VMIN`, minimalna liczba odebranych znaków zanim `read` powróci, i `VTIME`, czas oczekiwania dla funkcji `read`. Oba parametry są pamiętane w tablicy `c_cc`. To ważny temat, który omówimy szczegółowo w następnym podrozdziale. Na razie zauważ tyle, że następujący przykład pokazuje, jak wyłączyć znacznik `ICANON`:

```
#include <termios.h>

struct termios tdes;
.

.

tcgetattr(0, &tdes);
tdes.c_lflag |= ~ICANON;
tcsetattr(0, TCSAFLUSH, &tdes);
```

Jeśli znacznik **ISIG** został ustawiony, włączone jest przetwarzanie klawiszy przerwań *intr* i *quit*. Zwykle pozwala to użytkownikowi przerwać działający program. Jeśli znacznik **ISIG** nie został ustawiony, żadne sprawdzanie nie jest wykonywane i znaki *intr* i *quit* są przekazywane do programu czytającego bez zmian.

Jeśli znacznik **ECHO** został ustawiony, nie powinno cię dziwić, że znaki są powtarzane w miarę wprowadzania. Wyłączenie tego znacznika jest przydatne przy procedurach sprawdzania hasła, programach wykorzystujących klawisze do funkcji specjalnych jak przesuwanie kurSORA, itp.

Jeśli znacznik **ECHOE** jest ustawiony i znacznik **ECHO** też, znak kasowania jest powtarzany jako sekwencja znaków: cofanie-spacja-cofanie. Powoduje to wymazanie ostatniego znaku z ekranu terminala, dając użytkownikowi potwierdzenie, że znak został faktycznie skasowany. Jeśli znacznik **ECHOE** jest ustawiony, a znacznik **ECHO** nie, wtedy znak kasowania jest powtarzany jako sekwencja: spacja-cofanie i na terminalu typu CRT/VDU wymazuje znak znajdujący się na pozycji kurSora.

Jeśli znacznik **ECHONL** został ustawiony, znak nowego wiersza jest zawsze powtarzany, nawet jeśli inne znaczniki powtórzeń są wyłączone. Jest to przydatne przy terminalach, mających własne, lokalne powtarzanie (co często nazywa się trybem **półdupleksowym** (ang. *half-duplex*)).

Ostatnim wątkiem omówienia znacznikiem w tej grupie jest **NOFLSH**, który zabrania zwykłego opróżniania kolejki wejściowej i wyjściowej (gdy naciśnięty zostanie klawisz *intr* lub *quit*), oraz opróżniania kolejki wejściowej (gdy naciśnięty zostanie klawisz *susp*).

Ćwiczenie 9.3 Napisz program **ttystate**, drukujący bieżący stan terminala powiązanego ze standardowym wejściem. Wyjście powinno mieć formę wprowadzonych w tym podrozdziale nazw stałych preprocesora (na przykład **ICANON** i **ECHOE**). Sprawdź w podręczniku swojego systemu Unix pełną listę dostępnych nazw.

Ćwiczenie 9.4 Napisz program **ttyset**, pobierający wyjście z **ttystate** i ustawiający terminal powiązany z jego standardowym wyjściem w opisany stan. Czy programy **ttystate** i **ttyset** są w jakiś sposób przydatne, pojedynczo lub razem?

9.3.6 Parametry MIN i TIME

Parametry **MIN** i **TIME** mają znaczenie tylko wtedy, gdy znacznik **ICANON** jest wyłączone. Przeznaczone są one do dostrojenia sterowania przez program wprowadzaniem danych. Parametr **MIN** określa minimalną liczbę znaków, którą sterownik terminala musi odebrać, zanim powróci wywołanie funkcji **read** w odniesieniu do terminala. Parametr **TIME** określa wartość czasu oczekiwania (w dziesiątych częściach sekundy), pozwalającą na dodatkowy stopień kontroli.

Wartości parametrów **MIN** i **TIME** są pamiętane w tablicy **c_cc** struktury **termios**, która opisuje stan terminala. Ich pozycja w tablicy jest zdefiniowana przez stałe **VMIN** i **VTIME** w **<termios.h>**. Następujący fragment programu pokazuje, jak je ustawiać:

```
#include <termios.h>
struct termios tdes;
int ttyfd;

/* pobierz bieżący stan */
tcgetattr(ttyfd, &tdes);

tdes.c_lflag |= ~ICANON; /* wyłącz tryb kanoniczny */
tdes.c_cc[VMIN] = 64; /* w znakach */
tdes.c_cc[VTIME] = 2; /* dziesiąte części sekundy */

tcsetattr(0, TCSAFLUSH, &tdes);
```

Parametry **VMIN** i **VTIME** mają najczęściej te same wartości, co **VEOF** i **VEOL**. Oznacza to, że **MIN** i **TIME** zajmują to samo miejsce pamięci, co znaki **eof** i **eol**. Wynika stąd, że gdy przełączasz się z trybu kanonicznego na tryb niekanoniczny, musisz pamiętać o podaniu wartości **MIN** i **TIME**. W przeciwnym przypadku możesz uzyskać dziwne własności. (W szczególności jeśli twoim znakiem **eof** jest [Ctrl+D], twój program może czytać swoje wejście blokami po cztery znaki. Dlaczego?) Podobne argumenty dotyczą zmiany trybu w odwrotną stronę.

Istnieją cztery możliwe kombinacje **MIN** i **TIME**:

1. *Obydwa parametry MIN i TIME równe zeru.* W tym przypadku funkcja **read** zawsze będą powracać natychmiast. Jeśli znaki są obecne w kolejce wejściowej tego terminala (pamiętaj, że znak wejściowy może pojawić się w dowolnej chwili), będą umieszczone w buforze procesu. A więc, jeśli program wyłączając znacznik **ICANON** ustawia swój terminal sterujący w stan surowy, a oba parametry **MIN** i **TIME** są równe zeru, instrukcja:

```
nread = read(0, buffer, SOMESZ);
```

zwróci dowolną liczbę znaków, od zera do **SOMESZ**, w zależności od tego, ile znaków znajduje się w kolejce w chwili wywołania funkcji.

2. *Parametr MIN większy od zera, a TIME równy zeru.* Czas nie odgrywa tu żadnej roli. Funkcja **read** będzie zadowolona tylko wtedy, gdy na odczyt czeka **MIN** znaków. Zdarzy się tak nawet wtedy, gdy funkcja **read** żąda mniej niż **MIN** znaków.

Najbardziej trywialne ustawienie w tej kategorii to parametr MIN równy jeden, a TIME zero. Spowoduje to, że funkcja read powróci za każdym razem, gdy system otrzyma znak z linii terminala. Może to być przydatne; kiedy po prostu czytamy z klawiatury terminala, chociaż problem stanowią wtedy klawisze wysyłające sekwencje wielu znaków.

3. *Parametr MIN równy zeru, a TIME większy od zera.* W tym przypadku parametr MIN nie odgrywa żadnej roli. Czasomierz jest włączany w chwili wywołania funkcji read. Funkcja powraca, gdy tylko otrzyma pierwszy znak. Jeśli czas oczekiwania minie (kiedy uplynie TIME dziesiątych części sekundy), funkcja read zwróci zero znaków.
4. *Oba parametry MIN i TIME większe od zera.* Jest to prawdopodobnie najbardziej użyteczny i elastyczny przypadek. Czasomierz będzie aktywowany po otrzymaniu pierwszego znaku, a nie w chwili wywołania funkcji read. Jeśli MIN znaków zostanie otrzymany przed upływem czasu oczekiwania, funkcja read powraca. Jeśli czas oczekiwania zostanie osiągnięty, do użytkownika programu zwrócone zostaną znaki będące aktualnie w kolejce wejściowej. Ten tryb działania jest użyteczny, kiedy znaki wejściowe nadchodzą w pakietach wysyłanych w krótkim czasie. Upraszczają on programowanie i zmniejsza liczbę wywołań funkcji systemowych, które w przeciwnym przypadku byłyby wykonywane. Jest on przydatny na przykład przy obsłudze klawiszy funkcyjnych, które po naciśnięciu wysyłają serię znaków.

9.3.7 Inne funkcje systemowe terminala

Dodatkowe funkcje systemowe terminala pozwalają programistie na pewien stopień kontroli nad kolejką wejściową i wyjściową, utrzymywanymi przez sterownik terminala. Są one używane następująco:

Użycie

```
#include <termios.h>
int tcflush(int ttyfd, int queue);
int tcdrain(int ttyfd);
int tcflow(int ttyfd, int actions);
int tcsendbrk(int ttyfd, int duration);
```

Funkcja `tcflush` opróżnia określoną kolejkę. Jeśli parametr `queue` został ustalony na wartość `TCIFLUSH` (zdefiniowaną w `<termios.h>`), opróżniana jest kolejka wejściowa. Innymi słowy, wszystkie znaki z kolejki wejściowej są niszczone. Jeśli parametr `queue` ma wartość `TCOFLUSH`, opróżniana jest kolejka wyjściowa. Jeśli ma on natomiast wartość `TCIOFLUSH`, opróżniane są obie kolejki, wejściowa i wyjściowa.

Funkcja `tcdrain` powoduje, że proces czeka, aż całe bieżące wyjście zostanie zapisane do `ttyfd`.

Funkcja `tcflow` dostarcza kontroli start/stop sterownika terminala. Kiedy parametr `action` jest równy `TCOFF`, zawieszane zostaje wyjście. Wyjście może być wznowione za pomocą ponownego wywołania funkcji z wartością `action` ustawioną na `TCON`. Funkcja `tcflow` może być też użyta do wysłania do urządzenia terminala znaków `STOP` lub `START`, przez ustawienie parametru `action` odpowiednio na wartość `TCIOFF` lub `TCION`.

Funkcja `tcsendbrk` jest używana do wysłania przerwy, która odpowiada bitom zerowym o określonym przez `duration` czasie trwania. Jeśli parametr `duration` jest równy 0, wtedy bity będą wysyłane nie częściej niż co ćwierć sekundy i nie rzadziej niż co pół sekundy. Jeśli parametr `duration` jest różny od zera, wtedy czas wysyłania bitów będzie zależał od implementacji.

9.3.8 Sygnał zawieszenia

W rozdziale 6 widzieliśmy, że sygnał zawieszenia `SIGHUP` jest wysyłany do członków sesji, gdy kończy się proces wiodący sesję (pod warunkiem, że ma terminal sterujący). Sygnał ten może mieć także inne zastosowanie, przeznaczone dla środowiska, w którym połączenie pomiędzy komputerem i terminaliem może zostać przerwane, a sygnał nośnej związany z linią terminala zmniejszy się. Na przykład może się tak zdarzyć, gdy terminali są podłączone za pomocą linii telefonicznej lub pewnych lokalnych sieci komputerowych. W takich przypadkach sterownik terminala powinien wysłać sygnał `SIGHUP` do wszystkich procesów, które rozpoznają terminal jako swój terminal sterujący. Jeśli sygnał ten nie zostanie przechwycony, spowoduje zakończenie programu. (W odróżnieniu od `SIGINT`, sygnał `SIGHUP` zwykle zatrzymuje powłokę. W rzeczywistości użytkownik zostaje automatycznie wyrejestrowany z systemu, kiedy jego połączenie z systemem ulega przerwaniu – jest to ważna cecha bezpieczeństwa).

Zwykły programista powinien pozostawić sygnał `SIGHUP` w spokoju – służy on dobremu celowi. Możesz jednak chcieć przechwycić sygnał, aby wykonać jakieś gruntowne operacje czyszczące:

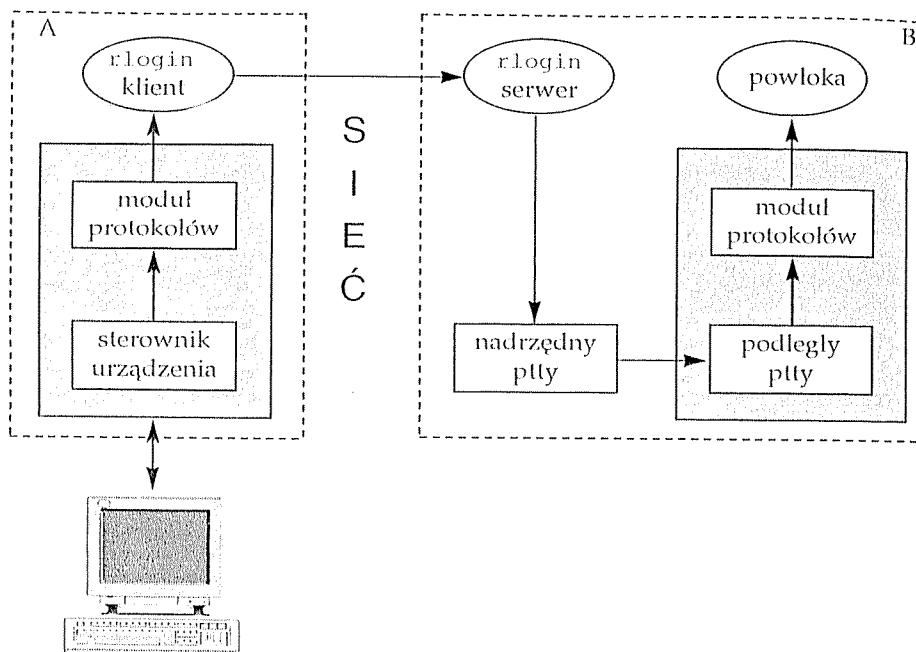
```
#include <signal.h>
void hup_action();
static struct sigaction act;

act.sa_handler=hup_action;
sigaction(SIGHUP, &act, NULL);
```

To podejście jest stosowane w niektórych edytorech, które zachowują edytowany plik i wysyłają pocztę przed wyjściem. Jeśli sygnał `SIGHUP` jest całkowicie ignorowany (przez ustawienie `act.sa_handler` na `SIG_IGN`), terminal zostanie zawieszony, kolejne wywołania funkcji `read` w odniesieniu do terminala będą zwracać 0, symulując koniec pliku.

9.4 Pseudoterminale

Inne powszechnie wykorzystanie struktury modulu protokołów polega na utworzeniu pseudoterminala w celu dostępu sieciowego. Pseudoterminal może być używany jako środek dołączenia terminala jednego komputera do powłoki innego, co pokazuje rysunek 9.4. Użytkownik jest tu połączony z komputerem A (klientem), ale używa powłoki komputera B (serwera). Strzałki na rysunku pokazują kierunek przekazywania znaków wejściowych z klawiatury. Uprościliśmy rysunek ignorując szczegóły stosu protokołu sieciowego w obu komputerach.



Rysunek 9.4 Zdalna rejestracja pomiędzy dwoma systemami Unixowymi

Kiedy użytkownik połączy się z powłoką na innym komputerze (najczęściej za pomocą polecenia `rlogin`), lokalne połączenie terminala zostanie zmodyfikowane. W miarę jak dane są czytane z lokalnego terminala, przechodzą niezmienione przez moduł protokołów, do procesu `rlogin` działającego na tym komputerze. Dlatego moduł protokołów na lokalnym komputerze jest ustawiony na pracę w surowym trybie przekazywania. Poniższy kod powinien przypomnieć, jak to należy zrobić:

```
#include <fcntl.h>
```

```
struct termios attr;
```

```
.
```

```
.
```

```
/* pobierz bieżący moduł protokołów */
tcgetattr(0, &attr);

/* czytaj po jednym znaku, bez czasu oczekiwania */
attr.c_cc[VMIN] = 1;
attr.c_cc[VTIME] = 0;
attr.c_lflag &= ~(ISIG|ECHO|ICANON);

/* ustaw nowy moduł protokołów */
tcsetattr(0, TCSAFLUSH, &attr);
```

Wtedy proces klienta `rlogin` przekazuje niezmienione dane przez sieć.

Kiedy serwer (B) otrzyma poczatkowe żądanie rejestracji, utworzy za pomocą wywołań `fork` i `exec` nową powłokę. Nowa powłoka nie ma żadnego związanego z nią terminala sterującego i dlatego budowany jest pseudoterminal, który udaje normalny sterownik urządzenia terminala. Pseudoterminal, zwykle nazywany **pseudo tty**, działa w bardzo podobny sposób co dwukierunkowy port, pozwalając po prostu dwóm różnym procesom na transfer danych. W naszym przykładzie łączy on proces powłoki z właściwym procesem sieciowym. Pseudoterminal stanowi para urządzeń, znana jako urządzenie nadziedzne (ang. *master*) i urządzenie podrzędne (ang. *slave*). Proces sieciowy otwiera, a następnie odczytuje i zapisuje do urządzenia nadziedzkiego, podczas gdy proces powłoki otwiera, a następnie zapisuje i odczytuje z urządzenia podrzędnego (za pomocą modułu protokołów). Dowolny zapis do urządzenia nadziedzkiego występuje jako wejście dla urządzenia podrzędnego i odwrotnie. Końcowy efekt jest taki, że użytkownikowi komputerowi-klientowi (A) wydaje się, że bezpośrednio używa powłoki, która aktualnie działa na serwerze (B). Podobnie, gdy dane mają być zapisane przez powłokę na serwerze, są przetwarzane przez moduł protokołów serwera (działający w trybie kanonicznym) i przekazywane bez zmiany do terminala klienta, bez modyfikowania przez moduł protokołów na komputerze-kliencie.

Chociaż sposób inicjacji pseudoterminali został ulepszony w nowych wersjach Uniksa i XSI, nadal nie jest zbyt wygodny! System Unix dostarcza ograniczonej liczby pseudoterminali i proces powłoki otwiera kolejny dostępny pseudoterminal. W systemie SVR4 jest to wykonywane przez otwarcie urządzenia `/dev/ptmx`, które określa i otwiera pierwsze nie używane urządzenie nadziedzne. Wszystkie urządzenia nadziedzne mają związane ze sobą urządzenia podrzędne. Aby przeszkodzić innym procesom w otwarciu związanego urządzenia podrzędnego, otwarcie `/dev/ptmx` blokuje także związane urządzenia podrzędne.

```
#include <fcntl.h>

int mfd;
.

.

/* otwórz pseudotermal -
 * uzyskaj deskryptor pliku dla urządzenia nadziedznego */
if (mfd = open("/dev/ptmx", O_RDWR)) == -1
{
```

```

    perror("Opening the master ppty");
    exit(1);
}
.
.
.
```

Przed otwarciem i odblokowaniem urządzenia podległego konieczne jest zapewnienie, że tylko proces z odpowiednimi prawami dostępu może odczytywać i zapisywać do niego. Funkcja grantpt zmienia tryb i właściciela urządzenia podległego na identyfikator efektywnego użytkownika związanego urządzenia nadległego. Funkcja unlockpt po prostu odblokowuje wewnętrzny znacznik stanu związany z urządzeniem podległym (to znaczy udostępnia je). Na koniec musimy otworzyć urządzenie podległe. Jednak na tym etapie nie znamy jego nazwy. Funkcja ptsname zwraca nazwę urządzenia podległego związanego z wyszczególnionym urządzeniem nadległym. Najczęściej urządzenie podległe ma nazwę w stylu /dev/pts/pettyxx. Następujący fragment kodu łączy wszystko w jedną całość:

```

#include <fcntl.h>

int mfd, sfd;
char *slavenm;
.

.

/* otwórz, jak przedtem, urządzenie nadległe */
if( (mfd = open("/dev/ptmx", O_RDWR)) == -1)
{
    perror("Opening the master ppty");
    exit(1);
}

/* zmień prawa dostępu urządzenia podległego */
if(grantpt(mfd)== -1)
{
    perror("Unable to grant access to ppty");
    exit(1);
}

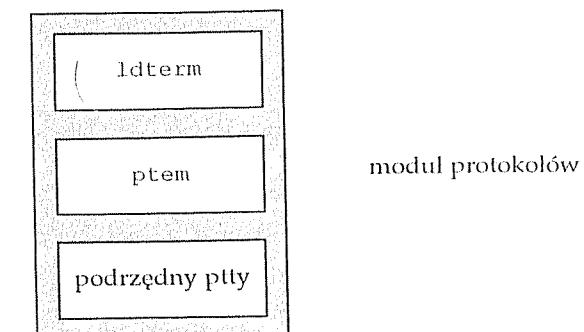
/* odblokuj związane z mfd urządzenie podległe */
if(unlockpt(mfd)== -1)
{
    perror("Unable to unlock the ppty");
    exit(1);
}

/* uzyskaj nazwę urządzenia podległego, a następnie otwórz je */
if( (slavenm = ptsname(mfd)) == NULL )
{
    perror("No slave name");
    exit(1);
}

if( (sfd = open(slavenm, O_RDWR)) == -1)
{
    perror("Opening slave ppty");
    exit(1);
}

```

Teraz, gdy uzyskałeś dostęp do sterownika urządzenia pseudoterminala, możemy zbudować zwiążany moduł protokołów. Aż do tej chwili traktowaliśmy moduł protokołów jako całość, podczas gdy w rzeczywistości składa się on z pewnej liczby wewnętrznych modułów jądra znanych jako STREAM (strumień). Standardowy moduł protokołów pseudoterminala STREAM składa się z trzech modułów: ldterm (modułu protokołów terminala), ptm (modułu emulacji pseudoterminala) oraz podległego końca pseudoterminala. Razem działają one jak prawdziwy terminal. Konfiguracja ta pokazana jest na rysunku 9.5.



Rysunek 9.5 STREAM dla urządzenia pseudoterminala

Aby zbudować ten STREAM, musimy umieścić dodatkowe moduły w urządzeniu podległym. Można tego dokonać za pomocą uniwersalnej funkcji ioctl. Na przykład:

```

/*
 * plik nagłówkowy stropts.h zawiera interfejs STREAMS i
 * definiuje makro I_PUSH używane jako drugi argument
 * ioctl()
 */
#include <stropts.h>

.

.

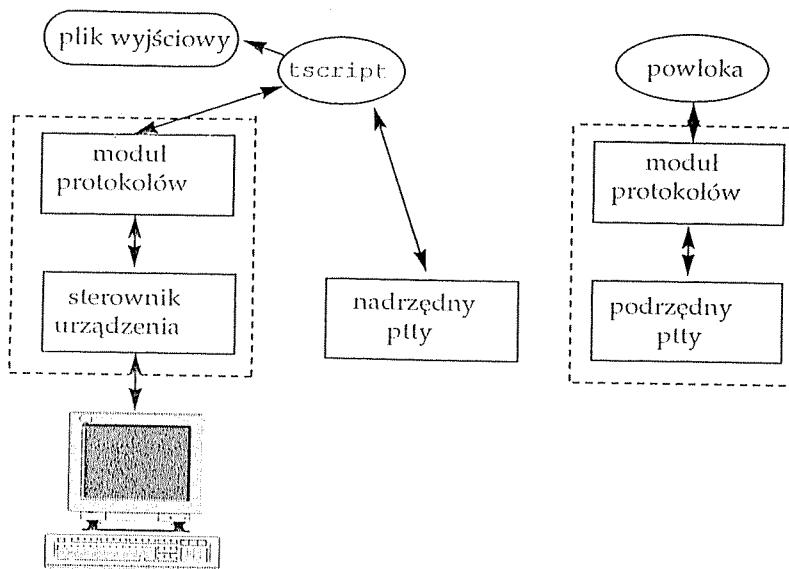
/* otwórz, jak poprzednio, urządzenie nadległe i podległe */
/* umieść dwa moduły w urządzeniu podległym */
ioctl(sfd, I_PUSH, "ptm");
ioctl(sfd, I_PUSH, "ldterm");

```

Teraz przejdziemy do bardziej rozbudowanego przykładu, o nazwie tscript, który używa pojęcia pseudoterminala na pojedynczym komputerze, aby przechwytywać wprowadzane znaki z interakcyjnej sesji powłoki bez wpływu na wykonanie tej sesji. Jest to podobne do polecenia Uniksa script. Prezentowane podejście może zostać rozszerzone przez sieć.

9.5 Przykład obsługi terminala: program tscript

Calkowity projekt jest zbudowany następująco: kiedy program tscript zostanie uruchomiony, za pomocą funkcji fork i exec utworzy powłokę użytkownika. Wszystkie dane zapisywane do terminala przez powłokę są bez wiedzy powłoki przechwytywane do pliku przez tscript, który zachowuje się tak, jakby miał całkowitą kontrolę nad modulem protokołów i tym samym nad terminaliem. Układ logiczny programu tscript został pokazany na rysunku 9.6.



Rysunek 9.6 Użycie pseudoterminalu w programie tscript

Głównymi elementami tego rozmieszczenia są:

tscript Pierwszy uruchomiony proces. Gdy pseudoterminal i moduł protokołów zostaną zainicjowane, używa funkcji fork i exec do utworzenia powłoki. Program tscript odgrywa teraz dwie role. Pierwszą jest czytanie z prawdziwego terminala i zapis wszystkich danych do urządzenia nadrukowego pseudoterminala. (Wszystkie dane zapisywane do nadrukowego pseudoterminala są przekazywane do podległego pseudoterminala). Drugą rolą programu tscript jest czytanie wyjścia z powłoki przez pseudoterminal i kopowanie do prawdziwego terminala i do pliku wyjściowego.

shell Przed uruchomieniem procesu shell (powłoki), do urządzenia podległego dokladane są moduły protokołów STREAM. Standardowe wejście powłoki, standardowe wyjście i standardowe wyjście komunikatów o błędach są wtedy podwajane, będąc podległym urządzeniem pseudoterminala.

Nasz przykład używa pliku nagłówkowego "tscript.h". Jego zawartość jest następująca:

```
/* tscript.h - plik nagłówkowy dla przykładu tscript */

#include <stropts.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stream.h>
#include <sys/ptms.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <termio.h>
#include <signal.h>
/* funkcje wewnętrzne */
void catch_child(int);
void runshell(int);
void script(int);
int ptyopen(int *, int *);

extern struct termios dattr;
```

Główny program tscript został pokazany poniżej. Pierwszym zadaniem programu jest ustawienie obsługi sygnalu SIGCHLD, a następnie otwarcie pseudoterminala. Program tworzy wtedy proces shell. Na koniec wywoływana jest procedura script, która czyta wejście z klawiatury, przekazywane do nadrukowego pseudoterminala, lub wejście z nadrukowego pseudoterminala, zapisywane do pliku wyjściowego, jak również przekazywane do standardowego wyjścia.

```
/* tscript - obsługa terminala */

#include "tscript.h"
struct termios dattr;

main()
{
    struct sigaction act;
    int mfd, sfd;
    char buf[512];

    /* zachowaj bieżące ustawienia terminala */
    tcgetattr(0, &dattr);

    /* otwórz pseudoterminal */
    if (ptyopen(&mfd, &sfd) == -1)
    {
        perror("opening pseudo tty");
        exit(1);
    }

    /* ustaw działanie po odebraniu SIGCHLD */
    act.sa_handler = catch_child;
    sigfillset(&(act.sa_mask));
```

```

sigaction(SIGCHLD, &act, NULL);

/* utwórz proces shell */
switch(fork()){
case -1: /* błąd */
    perror("fork failed on shell");
    exit(2);
case 0: /* potomek */
    close(mfd);
    runshell(sfd);
default: /* rodzic */
    close(sfd);
    script(mfd);
}

```

Powyższy program wywołuje cztery procedury. Pierwszą z nich jest `catch_child`, będąca procedurą obsługi sygnału `SIGCHLD`. Kiedy sygnał `SIGCHLD` zostanie odebrany, `catch_child` przywróci atrybuty terminala i zakończy się.

```

void catch_child(int signo)
{
    tcsetattr(0, TCSAFLUSH, &datr);
    exit(0);
}

```

Druga procedura, `pttopen`, otwiera pseudoterminal.

```

int pttopen(int *masterfd, int *slavefd)
{
    char *slavenm;

    /* otwórz pseudotermal - 
     * uzyskaj deskryptor pliku dla urządzenia nadzawanego */
    if( (*masterfd = open("/dev/ptmx", O_RDWR)) == -1)
        return (-1);

    /* zmień prawa dostępu urządzenia podzawanego */
    if(grantpt(*masterfd)== -1)
    {
        close(*masterfd);
        return (-1);
    }

    /* odblokuj związane z mfd urządzenie podzawne */
    if(unlockpt(*masterfd)== -1)
    {
        close(*masterfd);
        return(-1);
    }

    /* uzyskaj nazwę urządzenia podzawanego, a następnie otwórz je */
    if( (slavenm = ptsname(*masterfd)) == NULL )
    {

```

```

        close(*masterfd);
        return (-1);
    }

    if( (*slavefd = open(slavenm, O_RDWR)) == -1)
    {
        close(*masterfd);
        return(-1);
    }

    /* zbuduj moduł protokołów */
    if( ioctl(*slavefd, I_PUSH, "ptem") == -1)
    {
        close(*masterfd);
        close(*slavefd);
        return (-1);
    }
    if( ioctl(*slavefd, I_PUSH, "ldterm") == -1)
    {
        close(*masterfd);
        close(*slavefd);
        return (-1);
    }
    return (1);
}

```

Kolejną procedurą jest `runshell`. Wykonuje ona następujące zadania :

- wywołuje `setpgp`, aby powłoka działała we własnej grupie procesów. Pozwala to powłoce uzyskać pełną kontrolę nad obsługą sygnałów, szczególnie w odniesieniu do sterowania pracą.
- wywołuje funkcję systemową `dup2` w celu ustawienia `stdin`, `stdout` i `stderr` na referencję do podzawanego deskryptora pliku. Jest to krytyczny krok.
- za pomocą funkcji `exec` uruchamia powłokę, która działa aż do zakończenia przez użytkownika.

```

void runshell(int sfd)
{
    setpgp();

    dup2(sfd, 0);
    dup2(sfd, 1);
    dup2(sfd, 2);

    execl("/bin/sh", "sh", "-i", (char *)0);
}

```

Pierwszym zadaniem faktycznej procedury `script` jest zmiana modułu protokołów w ten sposób, aby działał w surowym trybie. Zostaje to osiągnięte za pomocą pobrania bieżących atrybutów, odpowiedniej ich zmiany i wywołania funkcji `tcsetattr`. Następnie procedura otwiera plik `output`. Później procedura używa funkcji

systemowej `select` (omawianej w rozdziale 7) w celu odpytywania wejść ze standardowego wejścia i nadziednego pseudoterminala. Jeśli informacja zostanie otrzymana ze standardowego wejścia, procedura `script` przekazuje ją bez zmian do nadziednego urządzenia pseudoterminala. Jeśli jednak wejście zostanie otrzymane z nadziednego terminala, procedura `script` zapisuje informację do terminala użytkownika i do pliku `output`.

```
void script(int mfd)
{
    int nread, ofile;
    fd_set set, master;
    struct termios attr;
    char buf[512];

    /* ustaw moduł protokołów w tryb surowy */
    tcgetattr(0, &attr);
    attr.c_cc[VMIN] = 1;
    attr.c_cc[VTIME] = 0;
    attr.c_lflag &= ~(ISIG|ECHO|ICANON);
    tcsetattr(0, TCSAFLUSH, &attr);

    /* otwórz plik output */
    ofile = open("output", O_CREAT|O_WRONLY|O_TRUNC, 0666);
    /* ustaw maskę bitową dla funkcji systemowej select */
    FD_ZERO(&master);
    FD_SET(0, &master);
    FD_SET(mfd, &master);

    /* funkcja select jest wywoływaną bez czasu oczekiwania,
     * powinna się zawiesić, dopóki zdarzenie nie nastąpi */
    while(set==master, select(mfd+1, &set, NULL, NULL, NULL) > 0)
    {
        /* sprawdź standardowe wejście */
        if(FD_ISSET(0, &set))
        {
            nread = read(0, buf, 512);
            write(mfd, buf, nread);
        }

        /* sprawdź urządzenie nadziedne */
        if(FD_ISSET(mfd, &set))
        {
            nread = read(mfd, buf, 512);
            write(ofile, buf, nread);
            write(1, buf, nread);
        }
    }
}
```

Następujące wyjście pokazuje, jak działa program `tscript`. Komentarze, oznaczone przez `#`, pokazują, która powłoka aktualnie działa.

```
$ ./tscript
```

```
$ ls -l tscript          # teraz uruchamiamy nową powłokę
-rwxr-xr-x 1 spate fcf 6984 Jan 22 21:57 tscript
```

```
$ head -2 /etc/passwd   # działa w nowej powłoce
root:x:0:1:0000-Admin(0000):/./bin/ksh
daemon:x:1:1:0000-Admin(0000):/:
```

```
$ exit                  # zakończenie nowej powłoki
```

```
$ cat output            # powracamy do znaku zachęty oryginalnej powłoki
-rwxr-xr-x 1 spate fcf 6984 Jan 22 21:57 tscript
root:x:0:1:0000-Admin(0000):/./bin/ksh
daemon:x:1:1:0000-Admin(0000):/:
```

Ćwiczenie 9.5 Dodaj do programu właściwą obsługę błędów i opcję umożliwiającą użytkownikowi wyszczególnienie alternatywnej nazwy pliku używanego jako plik wyjściowy. Jeśli żadna nazwa nie jest wyszczególniona, domyślnie użyj nazwy `output`.

Ćwiczenie 9.6 Równoważny standardowy program Uniksa `script` posiada opcję `-a`, która dodaje wyjście do pliku `output`. Zaimplementuj podobną opcję.

ROZDZIAŁ 10

Gniazda

- 10.1 Wprowadzenie
- 10.2 Rodzaje połączeń
- 10.3 Adresowanie
- 10.4 Interfejs gniazd
- 10.5 Programowanie modelu połączeniowego
- 10.6 Programowanie modelu bezpołączeniowego
- 10.7 Różnice między modelami

10.1 Wprowadzenie

W poprzednich rozdziałach omawialiśmy pewną liczbę mechanizmów komunikacji międzyprocesowej (IPC), które mogą być używane w systemie Unix. Jednak w nowoczesnym środowisku komputerowym użytkownicy i projektanci muszą stawić czola środowisku sieciowemu, zaprojektowanemu na podstawie architektury klient/serwer. Ta konfiguracja pozwala systemom wspólnie użytkować informacje i takie zasoby, jak pliki, przestrzeń dyskowa, procesor i urządzenia peryferyjne. Środowisko sieciowe klient/serwer nieuchronnie oznacza, że proces musi przekazywać informacje i współpracować z procesami występującymi na komputerze-klientce i na serwerze.

Z przyczyn historycznych tworzenie sieci w Uniksie poszło w dwóch kierunkach. Projektanci systemu Berkeley UNIX we wczesnych latach osiemdziesiątych opracowali swój sławny i powszechnie używany interfejs gniazd (ang. *sockets*), podczas gdy projektanci Systemu V w roku 1986 udostępnili swój interfejs warstwy transportowej (ang. *Transport Level Interface, TLI*). Podrozdział programowania sieci w dokumentacji X/Open często jest nazywany *XTI*. W XTI obsługuje się zarówno interfejs gniazd, jak i TLI. Podstawowa idea obu implementacji jest taka sama, ale TLI używa znacznie więcej struktur i jest o wiele bardziej skomplikowany niż interfejs gniazd. Dlatego w tym rozdziale skoncentrujemy się na dobrze znanym i wypróbowanym interfejsie gniazd. Gniazda dostarczają prostego interfejsu programistycznego, spójnego dla procesów na tym samym komputerze lub na różnych komputerach. Krótko mówiąc, cel gniazda stanowi dostarczenie sposobu komu-

nikacji międzyprocesowej, wystarczająco ogólnej, aby pozwolić na dwukierunkowe komunikaty między dwoma procesami niezależnie od tego, czy znajdują się one na tym samym komputerze, czy też na różnych komputerach.

Rozdział ten jest krótkim przeglądem podstawowych pojęć i urządzeń. Jeśli chcesz dowiedzieć się więcej, znajdziesz bardziej szczegółowe opracowania (na przykład Stevensa z 1992 roku) albo, jeszcze lepiej, zapytaj zaprzyjaźnionego eksperta oprogramowania sieciowego.

Zauważ, że zwykle będzie konieczne dołączenie specjalnych bibliotek (za pomocą dodania `-lxnet` lub innych znaczników do wiersza polecenia `cc`). Szczegóły znajdziesz w swoim podręczniku.

10.2 Rodzaje połączeń

Procesy, który potrzebują wysłać informację przez sieć, mogą wybrać jeden z dwóch sposobów komunikacji. **Model połączony** (ang. *connection oriented model*) lub **obwód wirtualny** (ang. *virtual circuit*) może być używany przez proces, który musi wysyłać niesformatowany, nieprzerwany strumień znaków do tego samego, stałego miejsca przeznaczenia; na przykład zdalne połaczenie rejestracyjne, w którym system klienta ma wirtualne połaczenie z serwerem. Jednak w niektórych przypadkach (na przykład, gdy serwer pragnie wysyłać do swoich klientów komunikat w trybie rozgłoszeniowym (ang. *broadcast*) i nie jest zbyt zainteresowany tym, którzy klienci odbiorą komunikat) proces może użyć modelu **bezpołączeniowego** (ang. *connectionless oriented model*). Tutaj proces wysyła komunikat do wyszczególnianego adresu sieciowego; może wysłać następny komunikat do innego adresu. Użyjmy metafory. Model połączony przypomina sieć telefoniczną. Model bezpołączenny przypomina trochę wysyłanie komunikatów w liście. W tym drugim przypadku nigdy nie jesteś absolutnie pewien, że twój komunikat dotarł, a jeśli chcesz uzyskać odpowiedź, musisz umieścić w liście swój adres. Model połączony jest dobry, kiedy potrzebujesz prawdziwego systemu interakcyjnego z określoną kolejnością komunikatów i potwierdzeń odbioru. Model bezpołączeniowy jest bardziej efektywny i przydatny w takich okolicznościach, jak wysyłanie komunikatów rozgłoszeniowych do wielu komputerów.

Przy każdej komunikacji między procesami na różnych komputerach, komputery klienta i serwera muszą być połączone: w warstwie sprzętowej za pomocą wyposażenia sieciowego (kable, karty) i urządzeń (ruterów) i w warstwie programowej za pomocą standardowego zestawu protokołów sieciowych. Protokół to po prostu zestaw reguł dotyczących wysyłania komunikatów między komputerami. Dlatego system Unix potrzebuje zestawu reguł dla modelu połączenniowego i modelu bezpołączenniowego. W modelu połączonym używamy protokołu sterowania transmisją (ang. *Transmission Control Protocol, TCP*), natomiast w modelu bezpołączennym protokołu datagramów użytkownika (ang. *User Datagram Protocol, UDP*). Datagram to inne określenie pakietu komunikatów.

10.3 Adresowanie

Kiedy procesy komunikują się przez sieć, musi istnieć mechanizm, dzięki któremu każdy proces będzie miał adres sieciowy komputera, na którym znajduje się inny proces. Zasadniczo adres podaje fizyczne rozmieszczenie komputera w sieci. Adresy są warstwowe, reprezentując różne warstwy sieci. Dalej skoncentrujemy się na tym, co jest niezbędne przy programowaniu za pomocą gniazd.

10.3.1 Adresowanie w Internecie

W sieciach światowych istnieje prawie powszechnie zaakceptowany standard adresowania – adresowanie internetowe (IP).

Adres IP składa się z czterech liczb dziesiętnych, rozdzielonych kropkami. Na przykład:

197.124.10.1

Te cztery liczby zawierają wystarczającą информацию do określenia położenia docelowej sieci, jak również samego komputera-hosta w tej sieci, stąd określenie *internet* – czyli sieć zawierająca inne sieci.

Systemowe funkcje sieciowe Uniksa nie potrafią obsługiwać adresów IP w formacie czterech liczb dziesiętnych. Na poziomie programowania adresy IP są przechowywane w typie `in_addr_t`. Programiści nie muszą martwić się o wewnętrzną reprezentację tego typu, ponieważ istnieje procedura o nazwie `inet_addr`, przekształcająca czteroczynkowy adres dziesiętny do typu `in_addr_t`.

Użycie

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *ip_address);
```

Procedura `inet_addr` pobiera adres IP w formie ciągu znaków w rodzaju "1.2.3.4" i zwraca adres internetowy w odpowiednim typie. Jeśli wywołanie nie powiedzie się, ponieważ ciąg znaków adresu IP ma nieprawidłowy format, zwrócona zostanie wartość (`in_addr_t`) -1.

Na przykład:

```
in_addr_t server;
server = inet_addr("197.124.10.1");
```

Jeśli proces chce odnieść się w późniejszych wywołaniach do adresu swojego komputera, plik nagłówkowy `<netinet/in.h>` definiuje stałą `INADDR_ANY` jako skrót adresu lokalnego hosta w formacie `in_addr_t`.

10.3.2 Porty

Program klienta musi wiedzieć, do którego komputera ma uzyskać dostęp oraz mieć możliwość połączenia się z właściwym procesem serwera. Proces serwera oczekuje połączeń na określonym numerze portu. Dlatego proces klienta powinien

prosić o połoczenie ze szczególnym komputerem i określonym portem (co jest równoważne umieszczeniu na kopercie oprócz adresu także numeru piętra lub mieszkania).

Niektóre numery są dobrze znany numerami portów i zapewniają zwyczajowo przyjęte usługi, na przykład `ftp` lub `rlogin`. Te numery są ustalone w pliku `/etc/services`. Zasadniczo numery portów mniejsze od 1024 są zarezerwowane dla procesów systemowych Uniksa. Wszystkie większe numery mogą być używane przez procesy użytkownika.

10.4 Interfejs gniazd

Istnieją standardowe struktury do przechowywania informacji o adresach i portach. Ogólna struktura adresu gniazda jest zdefiniowana w pliku nagłówkowym `<sys/socket.h>` jako:

```
struct sockaddr
{
    sa_family_t sa_family; /* rodzina adresów */
    char        sa_data[]; /* adres gniazda */
};
```

Jest to opisywane jako **gniazdo ogólne**, ponieważ w rzeczywistości używane są różne typy gniazd, zależnie od tego, czy gniazdo jest używane jako środek komunikacji międzymiędzyprosesowej (IPC) na tym samym komputerze uniksowym, czy jako punkt końcowy przy komunikacji procesów przez sieć. Specyficzna forma gniazda dla komunikacji sieciowej została pokazana poniżej:

```
#include <netinet/in.h>

struct sockaddr_in
{
    sa_family_t      sin_family; /* rodzina adresów internetowych */
    in_port_t        sin_port;   /* numer portu */
    struct in_addr   sin_addr;  /* przechowuje adres IP */
    unsigned char    sin_zero[8]; /* wypełnienie */
};
```

10.4.1 Tworzenie punktu końcowego transportu

We wszystkich formach komunikacji i klient, i serwer muszą ustalić swoje własne **punkty końcowe transportu** (ang. *transport end points*). Są one uchwytemi, używanymi do utworzenia połączenia przez sieć pomiędzy procesami. Ich tworzenie umożliwia funkcja systemowa `socket`:

Użycie

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Parametr `domain` mówi funkcji, gdzie ma być używane gniazdo. Na przykład `AF_INET` specyfikuje dla tworzenia sieci domenę internetu. Inną domeną, która

może być interesująca dla czytelnika, jest `AF_UNIX`, używana jeśli procesy znajdują się na tym samym komputerze.

Parametr `type` tworzonego gniazda określa, czy ma być ono używane w trybie połączniowym, czy też bezpołączniowym. Wartość `SOCK_STREAM` określa łącze nastawione na połączenia, a wartość `SOCK_DGRAM` łącze bezpołączniowe. Końcowy parametr, `protocol`, określa, który protokół powinien być używany przez to gniazdo. Zwykle jest on ustawiany na wartość 0, co oznacza, że domyślnie gniazdo `SOCK_STREAM` będzie używać protokołu TCP, a gniazdo `SOCK_DGRAM` protokołu UDP – standardowych protokołów Uniksa.

Funkcja systemowa `socket` normalnie zwraca nieujemną liczbę całkowitą, która jest deskryptorem pliku gniazda i umożliwia zastosowanie znanego modelu pliku uniksowego do gniazda.

10.5 Programowanie modelu połączniowego

Nadszedł już czas na rozpoczęcie nader ważnego przykładu. Demonstrując kilka ważniejszych funkcji systemowych opartych na gniazdach skoncentrujemy się na przykładzie, w którym klient wysyla swojemu serwerowi strumień znaków w postaci małych liter. Serwer zamienia je na wielkie litery i odsyła do klienta. W dalszych częściach tego rozdziału pokażemy ten sam przykład w trybie komunikacji bezpołączniowej.

Najpierw podamy zarys kodu dla procesu serwera:

```
/* proces serwera */

/* dołącz niezbędne pliki nagłówkowe */
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
    int sockfd;

    /* ustaw punkt końcowy transportu */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket call failed");
        exit(1);
    }

    /*
     * "zwiąż" adres serwera z punktem końcowym
     * zaczniętym nasłuchiwać przychodzących połączeń
     */

    pętla
        przyjmij połączenie
        utwórz proces potomny do pracy z połączeniem
        jeśli potomek
```

```

    ! wysyłaj i odbieraj informacje od klienta
}

Szablon procesu klienta wygląda następująco:

/* proces klienta */

/* dołącz niezbędne pliki nagłówkowe */
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

main()
{
    int sockfd;

    /* ustaw punkt końcowy transportu */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket call failed");
        exit(1);
    }

    /* połącz gniazdo z adresem serwera
       wysyłaj i odbieraj informacje od serwera */
}

```

Teraz zaczniemy wypełniać luki, zaczynając od logiki serwera.

10.5.1 Związywanie

Funkcja systemowa bind kojarzy prawdziwy adres sieciowy komputera z identyfikatorem gniazda.

Użycie

```

#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *address,
         size_t add_len);

```

Pierwszy parametr, sockfd, jest deskryptorem pliku gniazda zwróconym początkowo przez funkcję systemową socket. Drugi parametr jest tu określony jako wskaźnik do ogólnej struktury gniazda. Ponieważ jednak w naszym przykładzie wysyłamy informację przez sieć, w rzeczywistości dostarczymy adres одноśnej struct sockaddr_in, zawierającej informacje adresowe naszego serwera. Końcowy parametr zawiera wielkość faktycznie używanej struktury gniazda. Jeśli wywołanie funkcji bind jest pomyślne, zwraca 0. Przy błędzie wywołanie

bind zwróci -1, co może się zdarzyć, jeśli gniazdo dla danego adresu już istnieje. Wtedy zmieniąa errno będzie zawierać EADDRINUSE.

10.5.2 Nasłuchiwanie

Po związańiu, zanim dowolny system klienta będzie mógł połączyć się z niedawno utworzonym punktem końcowym serwera, serwer musi ustawić się w tryb czekania na połączenie. Wykonuje to za pomocą funkcji systemowej listen.

Użycie

```

#include <sys/socket.h>
int listen(int sockfd, int queue_size);

```

Parametr sockfd jest taki sam, jak poprzednio. Serwer może ustawić w kolejce do queue_size przybywających żądań połączenia. (XTL określa przenośne maksimum pięciu takich żądań).

10.5.3 Przyjmowanie

Kiedy serwer otrzyma od klienta żądanie connect, musi utworzyć całkowicie nowe gniazdo do obsługi określonej komunikacji. Pierwsze gniazdo jest używane tylko do nawiązania komunikacji. Tworzenie drugiego gniazda odbywa się za pomocą funkcji systemowej accept.

Użycie

```

#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *address,
           size_t *add_len);

```

Do funkcji systemowej accept przekazywany jest deskryptor nasłuchującego gniazda, zwrócony przez początkowe wywołanie funkcji systemowej socket. Po zakończeniu wywołania zwracana wartość jest identyfikatorem nowego gniazda używanego do komunikacji. Parametr address zostaje wypełniony informacją o kliencie. Jednak, ponieważ jest to połączenie ukierunkowane na komunikację, serwer najczęściej nie potrzebuje znać adresu klienta i dlatego parametr address może być zastąpiony przez NULL. Jeśli address jest różny od NULL, wtedy zmieniona wskazywana przez add_len powinna początkowo zawierać długość struktury adresu opisywanej przez parametr address. Po powrocie z wywołania funkcji *add_len będzie zawierał liczbę faktycznie skopiowanych bajtów.

Po objaśnieniu funkcji systemowych bind, listen i accept możemy uzupełnić kod serwera:

```

/* proces serwera */
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define SIZE sizeof(struct sockaddr_in)

int newsockfd;

main()
{
    int sockfd;

    /* zainicjuj gniazdo internetowe z numerem portu 7000
     * i adres lokalny, określony jako INADDR_ANY */
    struct sockaddr_in server = {AF_INET, 7000, INADDR_ANY};

    /* ustaw punkt końcowy transportu */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket call failed");
        exit(1);
    }

    /* zwiąż adres serwera z punktem końcowym */
    if ( bind(sockfd, (struct sockaddr *)&server, SIZE) == -1)
    {
        perror("bind call failed");
        exit(1);
    }

    /* zacznij nasłuchiwać przychodzących połączeń */
    if ( listen(sockfd, 5) == -1 )
    {
        perror("listen call failed");
        exit(1);
    }

    for ( ; ; )

        /* przyjmij połączenie */
        if ( (newsockfd = accept(sockfd, NULL, NULL)) == -1)
        {
            perror("accept call failed");
            continue;
        }

        /*
         * utwórz potomka do pracy z połączeniem
         * jeśli proces potomny
         * wysyłaj i odbieraj informacje od klienta
         */
    }
}

```

Kluczowym punktem jest tu użycie INADDR_ANY do reprezentacji lokalnego komputera.

Mamy teraz proces serwera zdolny do słuchania i przyjmowania nadchodzących połączeń. Zobaczmy teraz, jak klient może prosić o takie połączenie.

10.5.4 Przyłączenie klienta

Aby prosić o połączenie z procesem i komputerem serwera, klient używa trafię nazwanej funkcji systemowej connect:

Użycie

```

#include <sys/types.h>
#include <sys/socket.h>
int connect(int csockfd, const struct sockaddr *address,
            size_t add_len);

```

Pierwszy parametr, csockfd, jest deskryptorem pliku dla dołączanego gniazda klienta. Nie ma on żadnego związku z identyfikatorem gniazda na serwerze. Parametr address to wskaźnik do struktury zawierającej adres serwera i, ponownie, parametr add_len stanowi długość używanej struktury wyspecyfikowanego adresu.

Kontynuujmy nasz przykład. Kod klienta może być teraz rozszerzony następująco:

```

/* proces klienta */
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    struct sockaddr_in server = {AF_INET, 7000};

    /* przekształć i zapamiętaj adres IP serwera */
    server.sin_addr.s_addr = inet_addr("206.45.10.2");

    /* ustaw punkt końcowy transportu */
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket call failed");
        exit (1);
    }

    /* połącz gniazdo z adresem serwera */
    if ( connect(sockfd, (struct sockaddr *)&server, SIZE) == -1)
    {

```

```

    perror("connect call failed");
    exit(1);
}

/* wysyłaj i odbieraj informacje od serwera */
}

```

Musimy tu znać adres sieciowy komputera-serwera, używany w wywołaniu funkcji `inet_addr`. Zwykle dobrym miejscem do rozpoczęcia poszukiwań tego adresu dla twojej sieci będzie plik `/etc/hosts`.

10.5.5 Wysyłanie i odbieranie danych

Jeśli wszystko poszło dobrze, można teraz ustanowić obwód pomiędzy klientem i serwerem. Jeżeli gniazdo zostało ustawione na typ `SOCK_STREAM`, i klient, i serwer będą mieć (różne) deskryptory pliku, używane do odczytu lub zapisu. Najczęściej funkcje systemowe `read` i `write` mogą być używane w normalny sposób. Jednak istnieją dwie nowe funkcje systemowe, przydatne, jeśli trzeba ustawić dodatkowe opcje dotyczące sposobu przesyłania danych przez sieć. Funkcje `send` i `recv` są tak proste w użyciu jak `read` i `write`. W rzeczywistości, jeśli ich czwarty argument jest ustawiony na 0, zachowują się identycznie.

Użycie

```

#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buffer, size_t length,
             int flags);
ssize_t send(int sockfd, const void *buffer, size_t length,
             int flags);

```

Funkcja systemowa `recv` określa deskryptor pliku, z którego dane będą czytane, bufor, w którym powinny być umieszczone i wielkość bufora. Podobnie jak przy `read`, funkcja `recv` zwraca ilość odczytanych danych.

Parametr `flags` wpływa na sposób, w jaki dane mogą być otrzymywane. Możliwe wartości to:

<code>MSG_PEEK</code>	Proces może przejrzeć dane, bez faktycznego ich odbierania.
<code>MSG_OOB</code>	Zwykle dane są omijane i proces otrzymuje tylko dane pilne (ang. <i>out of band</i>), na przykład sygnał przerwania.
<code>MSG_WAITALL</code>	Wywołanie funkcji <code>recv</code> wróci tylko wtedy, gdy dostępna jest pełna ilość danych.

Jeśli parametr `flags` jest ustawiony na 0, funkcja `send` zachowuje się dokładnie jak `write`. Wysyła ona komunikat zawarty w `buffer` do lokalnego gniazda `sockfd`. Parametr `length` określa długość bufora `buffer`. Podobnie jak przy `recv`, parametr `flags` wpływa na sposób wysyłania komunikatów. Możliwe wartości to:

Rozdział 10: Gniazda

`MSG_OOB` Wyślij pilne (ang. *out of band*) dane.

`MSG_DONTROUTE` Komunikat będzie wysłany z pominięciem wszystkich warunków trasowania podstawowego protokołu. Zwykle oznacza to, że komunikat będzie wysłany trasą bezpośrednią, a nie najszybszą (najszybsza trasą może być dłuższa, w zależności od bieżącego obciążenia sieci).

Możemy teraz użyć tych funkcji w procesie serwera do konwersji odebranych komunikatów z małych liter na wielkie:

```

/* proces serwera */

main()
{
    /* zainicjuj gniazdo, jak poprzednio */

    char c;

    for (;;)
    {
        /* przyjmij połaczenie */
        if ( (newsockfd = accept(sockfd, NULL, NULL)) == -1)
        {
            perror("accept call failed");
            continue;
        }

        /* utwórz proces potomny do pracy z połączeniem */
        if ( fork() == 0)
        {
            /* odbierz dane */
            while (recv(newsockfd, &c, 1, 0) > 0)
            {
                /* zamień na wielkie litery i odeslij z powrotem */
                c = toupper(c);
                send(newsockfd, &c, 1, 0);
            }
        }
    }
}

```

Pamiętaj, że funkcja `fork` umożliwia naszemu serwerowi obsługę wielu klientów. Kod klienta powinien wyglądać następująco:

```

/* proces klienta */

main()
{
}

```

```

int sockfd;
char c, rc;

/* zainicjuj gniazdo i żądaj połączenia,
 * jak poprzednio */

/* wysyłaj i odbieraj informacje od serwera */
for(rc = '\n';;)
{
    if (rc == '\n')
        printf("Input a lower case character\n");
    c = getchar();
    send(sockfd, &c, 1, 0);
    recv(sockfd, &rc, 1, 0);
    printf("%c", rc);
}

```

10.5.6 Zamknięcie połączenia

Bardzo ważne jest rozsądne działanie przy nieoczekiwany zakončeniu procesu na drugim końcu gniazda. Ponieważ gniazdo jest mechanizmem komunikacji dwukierunkowej, nie można przewidzieć, czy proces będzie próbować odczytywać lub zapisywać, gdy zdarzy się przerwa w komunikacji. Trzeba wziąć pod uwagę obie możliwości.

Jeśli proces próbuje wysłać (za pomocą funkcji write lub send) dane do gniazda, które zostało odłączone, otrzyma sygnał SIGPIPE, który powinien być obsłużony w zwykły sposób, to znaczy za pomocą właściwego programu obsługi sygnału.

Jeśli funkcja read lub recv zwraca zero, wskazuje to na koniec pliku i koniec połączenia. Dlatego należy zawsze sprawdzać wartość zwracaną przez funkcje read lub recv i odpowiednio postępować.

Do gniazda może być użyta funkcja systemowa close. Jeśli używane jest gniazdo typu SOCK_STREAM, jądro gwarantuje, że wszystkie dane wyslane do gniazda zostaną wysłane do procesu odbiorczego. Może to spowodować wstrzymanie operacji close, aż wszystkie pozostałe dane zostaną dostarczone. (Jeśli gniazdo jest typu SOCK_DGRAM, wtedy zostaje zamknięte natychmiast).

Teraz możemy nareszcie uzyskać bardziej kompletny przykład procesów klienta i serwera przez dodanie obsługi sygnału do procesu serwera i instrukcji close do obu procesów. W naszym przykładzie nie jest to tak istotne, jak prostota przetwarzania. Jednak w prawdziwym środowisku klient/serwer te techniki powinny zapewnić dobrą obsługę wyjątków.

```
/* proces serwera */
```

```
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```

#include <signal.h>
#define SIZE sizeof(struct sockaddr_in)

void catcher(int sig);
int newsockfd;

main()
{
    int sockfd;
    char c;
    struct sockaddr_in server = {AF_INET, 7000, INADDR_ANY};
    static struct sigaction act;

    act.sa_handler = catcher;
    sigfillset(&(act.sa_mask));
    sigaction(SIGPIPE, &act, NULL);

    /* ustaw punkt końcowy transportu */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket call failed");
        exit(1);
    }

    /* zwiąż adres z punktem końcowym */
    if (bind(sockfd, (struct sockaddr *)&server, SIZE) == -1)
    {
        perror("bind call failed");
        exit(1);
    }

    /* zacznią nasłuchiwać przychodzących połączeń */
    if (listen(sockfd, 5) == -1)
    {
        perror("listen call failed");
        exit(1);
    }

    for (;;)
    {
        /* przyjmij połączenie */
        if ((newsockfd = accept(sockfd, NULL, NULL)) == -1)
        {
            perror("accept call failed");
            continue;
        }

        /* utwórz proces potomny do pracy z połączeniem */
        if (fork() == 0)
        {
            while (recv(newsockfd, &c, 1, 0) > 0)
            {
                c = toupper(c);
                send(newsockfd, &c, 1, 0)
            }
        }
    }
}

```

```

/* gdy klient nie wysyła dłużej informacji,
   gniazdo może być zamknięte i proces
   potomny zakończony */
close(newsockfd);
exit(0);
}

/* rodzic nie potrzebuje newsockfd */
close(newsockfd);

}
}

void catcher(int sig)
{
    close(newsockfd);
    exit(0);
}

```

A oto kod procesu klienta:

```

/* proces klienta */
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    char c, rc;
    struct sockaddr_in server = {AF_INET, 7000};

    /* przekształć i zapamiętaj adres IP serwera */
    server.sin_addr.s_addr = inet_addr("197.45.10.2");

    /* ustaw punkt końcowy transportu */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket call failed");
        exit(1);
    }

    /* połącz gniazdo z adresem serwera */
    if (connect(sockfd, (struct sockaddr *)&server, SIZE) == -1)
    {
        perror("connect call failed");
        exit(1);
    }

    /* wysyłaj i odbieraj informacje od serwera */
    for(rc = '\n'; ;)
    {
        if (rc == '\n')
            printf("Input a lower case character\n");

```

```

c = getchar();
send(sockfd, &c, 1, 0);

if(recv(sockfd, &rc, 1, 0)>0)
    printf("%c", rc);
else
{
    printf("server has died\n");
    close(sockfd);
    exit(1);
}
}

```

Ćwiczenie 10.1 Uruchom podany kod z więcej niż jednym klientem. Co się stanie, gdy wszystkie procesy klienta się zakończą?

Ćwiczenie 10.2 Dostosuj kod tak, że jeśli wszystkie procesy klienta zakończą się i nie będzie żądań nowych połączeń, serwer po odpowiednim czasie przekroczy czas oczekiwania.

Ćwiczenie 10.3 Dostosuj kod w ten sposób, aby dwa komunikujące się procesy mogły znajdować się na tym samym komputerze. Tym razem gniazdo musieć mieć typ AF_UNIX.

10.6 Programowanie modelu bezpołączniowego

Teraz zmodyfikujemy nasz przykład, używając modelu bezpołączniowego. Główna różnica pomiędzy modelem połączniowym a bezpośrednim sprowadza się do tego, że w trybie bezpołączniowym pakiety transmitowane pomiędzy klientem i serwerem będą przychodzić do miejsca przeznaczenia w nieokreślonej kolejności. Z punktu widzenia programowania w modelu bezpołączniowym proces pragnący wysyłać lub otrzymywać komunikaty przez sieć musi utworzyć swoje własne lokalne gniazdo i związać z nim za pomocą funkcji bind swój własny adres sieciowy. Proces może wtedy skutecznie używać tego gniazda jako bramy (ang. gateway) do sieci. Aby wysłać komunikat, proces musi znać adres przeznaczenia; może to być adres rozgłoszeniowy dotyczący wielu komputerów.

10.6.1 Wysyłanie i odbieranie komunikatów

Jednymi nowymi funkcjami systemowymi w modelu bezpołączniowym są funkcje `sendto` i `recvfrom`.

Parametr `sockfd` w obu funkcjach określa lokalnie związane gniazdo, przez które komunikaty będą wysyłane i odbierane.

Jeśli parametr `sendto` został ustawiony na `NULL`, funkcja `recvfrom` działa dokładnie w taki sam sposób jak funkcja `recv`. Wskaźnik `message` jest buforem, w którym są umieszczane odebrane komunikaty, a `length` liczbą bajtów odczytyanych do bufora `message`. Parametr `flags` przybiera identyczne wartości jak

w funkcji recv. Ostatnie dwa parametry pomagają w bezpołączeniowej formie komunikacji. Struktura send_addr jest zapelniana informacją adresową komputera, który wysłał komunikat. Oznacza to, że proces odbierający, jeśli chce, może wysłać odpowiedź. Końcowy parametr stanowi wskaźnik do liczby całkowitej size_t, która, po zakończeniu wywołania, będzie zawierała długość adresu.

Użycie

```
ssize_t recvfrom(int sockfd, void *message, size_t length,
                 int flags, struct sockaddr *send_addr,
                 size_t *add_len);
ssize_t sendto(int sockfd, const void *message, size_t length,
               int flags, const struct sockaddr *dest_addr,
               size_t dest_len);
```

Funkcja sendto stanowi przeciwnieństwo funkcji recvfrom. Tym razem parametr dest_addr określa adres równorzędny dla wysyłanego komunikatu, a dest_len określa długość adresu.

Poniżej zamieszczamy nasz przykład dla modelu bezpołączeniowego:

```
/* serwer */
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    char c;

    /* lokalny port serwera */
    struct sockaddr_in server = {AF_INET, 7000, INADDR_ANY};

    /* struktura do umieszczenia adresu procesu 2 */
    struct sockaddr_in client;
    int client_len = SIZE;

    /* ustaw punkt końcowy transportu */
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        perror("socket call failed");
        exit(1);
    }

    /* zwiąż adres lokalny z punktem końcowym */
    if ((bind(sockfd, (struct sockaddr *)&server, SIZE) == -1))
    {
        perror("bind call failed");
    }
}
```

```
exit(1);
}
/* czekaj w pętli na komunikaty */
for( ; );
{
    /* odbierz komunikat i zapamiętuj adres
       klienta */
    if(recvfrom(sockfd, &c, 1, 0,
                &client, &client_len) == -1)
    {
        perror("server: receiving");
        continue;
    }

    c = toupper(c);

    /* odeslij komunikat z powrotem do adresata */
    if (sendto(sockfd, &c, 1, 0, &client, client_len) == -1)
    {
        perror("server: sending");
        continue;
    }
}
```

I kod dla klienta :

```
/* proces klienta */
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SIZE sizeof(struct sockaddr_in)

main()
{
    int sockfd;
    char c;

    /* lokalny port klienta */
    struct sockaddr_in client = {AF_INET, INADDR_ANY, INADDR_ANY};

    /* zdalny adres serwera */
    struct sockaddr_in server = {AF_INET, 7000};
    /* przekształć i zapamiętaj adres IP serwera */
    server.sin_addr.s_addr = inet_addr("197.45.10.2");

    /* ustaw punkt końcowy transportu */
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        perror("socket call failed");
        exit(1);
    }
```

```

/* zwiąż lokalny adres z punktem końcowym */
if ( bind(sockfd, (struct sockaddr *)&client, sizeof(client)) == -1)
{
    perror("bind call failed");
    exit(1);
}

/* odczytaj znak z klawiatury */
while( read(0, &c, 1) != 0)
{

    /* wyślij znak do serwera */
    if( sendto(sockfd, &c, 1, 0, &server, sizeof(server)) == -1)
    {
        perror("client: sending");
        continue;
    }

    /* odbierz z powrotem komunikat */
    if(recv(sockfd, &c, 1, 0)== -1)
    {
        perror("client: receiving");
        continue;
    }

    write(1, &c, 1);
}

```

Ćwiczenie 10.4 Uruchom powyższy kod z pewną liczbą klientów. W jaki sposób serwer może wybrać, od którego klienta odbierać dane?

10.7 Różnice między modelami

Warto zastanowić się nad różnicami między tymi dwoma przykładami z punktu widzenia programowania.

W obu modelach serwer musi utworzyć gniazdo i związać z nim swój lokalny adres. W modelu połączonym serwer musi wtedy zacząć nasłuchiwać przybywających połączeń, co nie jest konieczne w scenariuszu bezpołączonym, ponieważ klient ma tu więcej pracy.

Z perspektywy klienta w modelu połączonym klient tylko łączy się z serwerem. W modelu bezpołączonym klient musi utworzyć gniazdo i związać z tym gniazdem swój lokalny adres.

Na koniec, inne funkcje systemowe są wykorzystywane do transmisji danych. Funkcje systemowe `send` i `recv` mogą być używane w obu modelach. Jednak w modelu bezpołączonym zwykle używane są funkcje `sendto` i `recvfrom`, więc serwer może uzyskać informację o nadawcy komunikatu i stosownie odpowiedzieć.

ROZDZIAŁ 11

Standardowa Biblioteka I/O

- 11.1 Wprowadzenie
- 11.2 Struktura FILE
- 11.3 Otwieranie i zamykanie strumieni plików: `fopen` i `fclose`
- 11.4 Operacje I/O dla pojedynczego znaku: `getc` i `putc`
- 11.5 Zwracanie znaku do strumienia pliku: `ungetc`
- 11.6 Standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach
- 11.7 Standardowe procedury stanu I/O
- 11.8 Wejście i wyjście linii
- 11.9 Wejście i wyjście binarne: `fread` i `fwrite`
- 11.10 Bezpośredni dostęp do pliku: `fseek`, `rewind` i `ftell`
- 11.11 Formatowane wyjście: rodzina funkcji `printf`
- 11.12 Formatowane wejście: rodzina funkcji `scanf`
- 11.13 Uruchamianie programów za pomocą Standardowej Biblioteki I/O
- 11.14 Rozmaite funkcje

11.1 Wprowadzenie

W końcowych rozdziałach książki omówimy kilka z bibliotek podprogramów standardowych, dostarczanych przez Uniksa (i z różnym stopniem kompletności przez wiele systemów kompilatorów C w innych środowiskach).

Zaczniemy od wyjątkowo ważnej Standardowej Bibliotece I/O, która stanowi większość części biblioteki C dostarczanej ze wszystkimi systemami Uniksa. Standardowe I/O skrótnie przedstawiliśmy w rozdziale 2; poza tym spotykaleś już kilka jego procedur, na przykład `getchar` i `printf`.

Głównym celem Standardowej Biblioteki I/O jest zapewnienie efektywnych, obszernych i przenośnych ułatwień dostępu do pliku. Procedury, które wchodzą w skład biblioteki, osiągają efektywność za pomocą niewidocznego dla użytkownika mechanizmu automatycznego buforowania, zmniejszającego liczbę rzeczy-

wistych dostępów do pliku i liczbę wykonywanych wywołań funkcji systemowych niskiego poziomu. Biblioteka jest obszerna, ponieważ oferuje dużo więcej ułatwień (takich jak formatowane wyjście i konwersja danych), niż pierwotne funkcje dostępu do pliku `read` i `write`. Standardowe procedury I/O są przenośne, ponieważ nie zostały związane z żadną specyficzną cechą systemu Unix, a nawet stały się częścią niezależnego od Uniksa standardu ANSI dla języka C. Każdy wartościowy kompilator C oferuje dostęp do pełnej implementacji Standardowej Biblioteki I/O, niezależnie od systemu operacyjnego hosta.

11.2 Struktura FILE

Standardowe procedury I/O identyfikują plik za pomocą wskaźnika do struktury typu `FILE`. Gdy wywoływana jest większość procedur Standardowego I/O, jeden z przekazywanych parametrów stanowi wskaźnik do struktury `FILE`, wskazującej, który plik wejściowy albo wyjściowy ma być używany. Dlatego wskaźnik do struktury `FILE` może być porównywany do całkowitych deskryptorów pliku używanych przy funkcjach `read`, `write`, itd.

Definicję struktury `FILE` można znaleźć w standardowym pliku nagłówkowym `<stdio.h>`. Trzeba podkreślić, że programista bardzo rzadko będzie zainteresowany rzeczywistą implementacją typu `FILE`. Definicja tej struktury zmienia się w zależności od systemu.

Wszystkie dane odczytywane lub zapisywane do pliku przechodzą przez bufor znaków struktury `FILE`. Na przykład procedura wyjściowa ze Standardowej Biblioteki I/O zapiełnia bufor znak po znaku. Jeśli bufor zostanie zapelniony, wewnętrzna procedura Standardowego I/O zapisuje jego zawartość do pliku za pomocą funkcji systemowej `write`. Jest to niewidoczne dla programu użytkownika. Wielkość bufora wynosi `BUFSIZE` bajtów. Stala `BUFSIZE` została zdefiniowana w `<stdio.h>` i, jak widzieliśmy w rozdziale 2, zwykle jest równa współczynnikowi blokowania dysku w środowisku hosta. Typowe wartości to 512 i 1024, albo większe.

Podobnie procedura wejściowa pobiera dane z bufora powiązanego ze strukturą `FILE`. Jeśli bufor zostanie opróżniony, odczytem z pliku za pomocą funkcji systemowej `read` zostanie zapelniony inny bufor. I znów ta operacja będzie niewidoczna dla programu użytkownika.

Mechanizm buforowania Standardowej Biblioteki I/O zapewnia, że dane zawsze są odczytywane lub zapisywane do zewnętrznego pliku w kawałkach o typowej wielkości. W rezultacie liczba dostępów do pliku i wewnętrznych wywołań funkcji systemowych jest utrzymywana na poziomie optymalnym. Jednak, ponieważ to buforowanie wykonują wewnętrznie same procedury Standardowego I/O, programista może używać procedur z biblioteki, logicznie odczytujących lub zapisujących dowolną liczbę bajtów, nawet po jednym. Dlatego program może być pisany w sposób odzwierciedlający strukturę problemu, a sprawy efektywności w większości można pozostawić bibliotece. Jak zobaczymy, procedury biblioteki standarowej dostarczają prostych w użyciu własności formatujących. Z tego powodu Standardowe I/O jest preferowaną metodą dostępu do pliku w wielu zastosowaniach.

11.3 Otwieranie i zamykanie strumieni plików: `fopen` i `fclose`

Użycie

```
#include <stdio.h>
FILE * fopen(const char *filename, const char *type);
int fclose(FILE *stream);
```

Procedury `fopen` i `fclose` są pochodząymi ze Standardowej Biblioteki I/O ekwiwalentami funkcji `open` i `close`. Procedura `fopen` otwiera plik identyfikowany przez `filename` i kojarzy z nim wskaźnik do struktury `FILE`. Jeśli procedura `fopen` została wykonana pomyślnie, zwraca wskaźnik do struktury `FILE`, identyfikujący otwarty plik (w rzeczywistości wskazywana struktura `FILE` jest składową utrzymywanej wewnętrznie tablicy). Procedura `fclose` zamyka plik identyfikowany przez `stream`, i jeśli jest używana dla wyjścia, opróżnia wszystkie dane pozostające w wewnętrznym buforze.

Jeśli procedura `fopen` nie powiedzie się, zwróci stałą `NULL`, która oznacza pusty wskaźnik i jest zdefiniowana w `<stdio.h>`. Wtedy zewnętrzna zmenna całkowita błędu `errno` będzie, podobnie jak przy `open`, zawierać kod wskazujący przyczynę błędu.

Drugi parametr `fopen` wskazuje napis, który określa tryb dostępu. Może on przybierać następujące podstawowe wartości:

- r otwórz `filename` tylko do odczytu. (Jeśli plik nie istnieje, wywołanie zawiedzie i `fopen` zwróci `NULL`).
- w utwórz lub obetnij `filename`, i otwórz go tylko do zapisu
- a otwórz `filename` tylko do zapisu; wszystkie zapisywane dane będą automatycznie dodawane do końca pliku. Jeśli plik nie istnieje, wtedy utwórz go do zapisu.

Plik może być też otwarty do aktualizacji, co w tym kontekście oznacza, że program odczytuje i zapisuje do pliku. Innymi słowy, program może mieszać operacje wejściowe i wyjściowe dla tego samego pliku, bez ponownego otwierania go. Jednak jest to bardziej restrykcyjne niż tryb odczytu/zapisu obsługiwany przez `read` i `write`, na skutek mechanizmu buforowania Standardowej Biblioteki I/O. W szczególności operacja wejściowa nie może wystąpić po wyjściowej, dopóki nie będzie wykonane interwencyjne wywołanie jednej z dwóch procedur Standardowego I/O: `fseek` lub `rewind`. Procedury te dostosowują wewnętrznie utrzymywany wskaźnik odczytu-zapisu (omówimy je dalej). Podobnie operacja wyjściowa nie może wystąpić po wejściowej bez uprzedniego wywołania procedury `fseek` lub `rewind` lub też procedury wejściowej, która pozycjonuje program na koniec pliku. Tryb aktualizacji jest wskazywany przez dodatkowy symbol „+” w typie argumentu przekazywanego do `fopen`. Możemy modyfikować w ten sposób wszystkie trzy spotkane powyżej napisy:

- r+ otwórz filename do odczytu i zapisu. I znów fopen zawiedzie, jeśli plik nie istnieje.
- w+ utwórz lub obetnij filename i otwórz do odczytu i zapisu
- a+ otwórz do odczytu i zapisu. Przy zapisie dane będą dodawane do końca pliku. Jeśli plik nie istnieje, wtedy będzie utworzony do zapisu.

W kilku środowiskach do r, w lub a trzeba dodać także 'b' w celu uzyskania dostępu do plików binarnych, a nie tekstowych. Na przykład: rb.

Kiedy fopen tworzy plik, zwykle nadaje mu prawa dostępu ustawione na wartość 0666. Umożliwia to wszystkim użytkownikom odczytywanie i zapisywanie pliku. Te domyślne prawa dostępu mogą być zmienione za pomocą ustawienia wartości umask dla procesu na wartość niezerową. (Funkcję systemową umask omawialiśmy w rozdziale 3).

Następujący zarys programu demonstruje użycie procedury fopen i jej związek z fclose. Przykład ten powoduje otwarcie pliku indata do odczytu, pod warunkiem że on istnieje, oraz utworzenie lub obcięcie pliku outdata. Procedurę obsługi błędów fatal znamy już z poprzednich rozdziałów. Przekazuje ona po prostu swój argument napisowy do perror, a następnie wywołuje exit w celu zakończenia działania.

```
#include <stdio.h>

char *inname = "indata";
char *outname = "outdata";

main() {
    FILE *inf, *outf;

    if( (inf = fopen(inname, "r")) == NULL)
        fatal("Could not open input file");

    if( (outf = fopen(outname, "w")) == NULL)
        fatal("Could not open output file");

    /* wykonaj coś pozytecznego .... */

    fclose(inf);
    fclose(outf);

    exit(0);
}
```

Właściwie w tym konkretnym kontekście nie jest potrzebne żadne wywołanie procedury fclose. Deskryptory pliku skojarzone z inf i outf będą automatycznie zamknięte, gdy proces się zakończy, a funkcja exit automatycznie opróżni dane pozostające w buforze skojarzonym z outf, zapisując je do pliku outdata.

Procedurą blisko związaną z fclose jest fflush:

Powoduje ona opróżnienie bufora wyjściowego skojarzonego ze stream; innymi słowy, dane z bufora są natychmiast zapisywane do pliku, bez względu na to, czy bufor pliku jest pełny, czy też nie. Zapewnia to zgodność zewnętrznego pliku

z widzianą przez proces rzeczywistością. (Pamiętaj, że jeśli proces jest tym zainteresowany, dane z bufora mogą już być zapisane do pliku. Mechanizm buforowania jest przezroczysty). Dowolne dane wejściowe są niszczone.

Użycie

```
#include <stdio.h>

int fflush(FILE *stream);
```

Po wywołaniu procedury fflush strumień stream pozostaje otwarty. Podobnie jak fclose, fflush zwraca stałą EOF w przypadku błędu, a zero w przypadku powodzenia. (Stała EOF jest zdefiniowana w <stdio.h> jako -1. Właściwie oznacza ona koniec pliku, ale może też być używana do wskazywania błędów).

11.4 Operacje I/O dla pojedynczego znaku: getc i putc

Użycie

```
#include <stdio.h>
int getc(FILE *inf);
int putc(int c, FILE *outf);
```

Najprostszymi procedurami wejścia i wyjścia dostarczonymi przez Standardową Bibliotekę I/O są getc i putc. Procedura getc zwraca następny znak ze strumienia wejściowego inf. Procedura putc umieszcza znak, wskazywany tu przez c, w strumieniu wyjściowym outf.

Dla obu procedur znak c jest zdefiniowany, by móc niezgodnie z intuicją, jako typ int, a nie char. Umożliwia to używanie procedury z zestawem znaków o szerokości 16 bitów. Pozwala także procedurze getc zwracać stałą EOF, która, ponieważ przybiera wartość -1, znajduje się poza możliwym zakresem wartości dla zmiennej typu unsigned char. Stała EOF jest używana przez getc do wskazania, że osiągnięto koniec pliku albo wystąpił błąd. Procedura putc też może zwrócić stałą EOF w przypadku błędu.

Następujący przykład jest nową wersją procedury copyfile, którą wprowadziliśmy w rozdziale 2; zamiast użycia funkcji read i write, używamy tu getc i putc:

```
#include <stdio.h>

/* kopiuje plik f1 do f2 używając Standardowego I/O */
int copyfile(const char *f1, const char *f2)
{
    FILE *inf, *outf;
```

```

int c;

if( (inf = fopen(f1, "r")) == NULL)
    return (-1);

if( (outf = fopen(f2, "w")) == NULL)
{
    fclose(inf);
    return (-2);
}

while( (c = getc(inf)) != EOF)
    putc(c, outf);

fclose(inf);
fclose(outf);
return (0);
}

```

Podstawowa forma wewnętrznej pętli `while` jest prawdopodobnie bliska temu, co w języku C określamy komunalem. I znów zwróć uwagę, że zmienna `c` została zdefiniowana jako `int`, a nie jako `char`.

I jeszcze małe ostrzeżenie: `getc` i `putc` mogą nie być funkcjami. Mogą to być makroinstrukcje zdefiniowane w `<stdio.h>`, rozwijane w linii. Jako makroinstrukcje `getc` i `putc` nie będą zachowywać się rozsądnie, jeśli dostaną argumenty z efektami ubocznymi. W szczególności wyrażenia `getc(*f++)` i `putc(c, *f++)` nie będą dawać prawidłowych wyników. Oczywiście, implementacja w postaci makroinstrukcji zapewnia efektywność przez wyeliminowanie niepotrzebnych wywołań funkcji. Jednak puryści będą zadowoleni wiedząc, że `fgetc` i `fputc` są prawdziwymi funkcjami, które wykonują to samo, co ich imiennicy. Są one rzadko używane, ale przydatne, jeśli nazwa funkcji musi być przekazana jako parametr do innej funkcji.

Ćwiczenie 11.1 W ćwiczeniach 2.4 i 2.5 opisywaliśmy program o nazwie `count`, który wyświetlał liczbę znaków, słów i linii w pliku wejściowym. (Pamiętaj, że słowo jest zdefiniowane jako dowolny pojedynczy znak niealfanumeryczny albo dowolna ciągła sekwencja znaków alfanumerycznych). Przepisz program `count` używając `getc`.

Ćwiczenie 11.2 Używając `getc`, napisz program, który rejestruje rozkład znaków w pliku, to znaczy rejestruje, ile razy każdy ze znaków oddziennie wystąpił w pliku. Jeden ze sposobów wykonania tego polega na deklaracji tablicy liczb całkowitych typu `long` działającej jako licznik, a następnie wykorzystaniu wartości całkowitej każdego znaku wejściowego jako indeksu tej tablicy. W tym przypadku upewnij się, że twój program zachowuje się rozsądnie nawet, jeśli na twoim komputerze typ `char` jest domyślnie zdefiniowany jako `signed` (co oznacza, że bajty wejściowe mogą przybierać wartości ujemne). Używając `printf` i `putc` spraw, żeby twój program wyświetlał prosty histogram odnalezionej rozkładu.

11.5 Zwracanie znaku do strumienia pliku: `ungetc`

Użycie

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Procedura `ungetc` wstawia znak `c` z powrotem do strumienia `stream`. Jest to tylko operacja logiczna. Sam plik wejściowy nie zostanie zmieniony. Jeśli wywołanie `ungetc` będzie pomyślne, znak zawarty w `c` stanie się następnym znakiem odczytanym przez `getc`. Gwarantowany jest zwrot tylko jednego znaku. Jeśli próba wstawiania `c` zawiedzie, `ungetc` zwraca wartość `EOF`. Próba zwrócenia samego znaku `EOF` zawsze zawodzi. Jednak nie jest to zwykle problemem, ponieważ kolejne wywołania `getc` po osiągnięciu końca pliku zawsze będą się kończyć zwracaniem `EOF`.

Najczęściej funkcja `ungetc` jest używana do przywracenia strumienia wejściowego do stanu oryginalnego po odczytaniu w celach testowych jednego znaku za dużo. Następująca procedura `getword` wykorzystuje tę prostą technikę do zwrócenia napisu, który zawiera dowolną ciągłą sekwencję znaków alfanumerycznych albo pojedynczy znak niealfanumeryczny. Koniec pliku jest wskazywany przez zwrot wartości `NULL`. Procedura `getword` pobiera jako argument wskaźnik `FILE`. Używa ona dwóch testowych makroinstrukcji ze standardowego pliku nagłówkowego `<ctype.h>`. Pierwszą z nich jest `isspace`, która określa, czy znak jest znakiem odstępu (białej spacji), takim jak sama spacja, tabulacja lub znak nowego wiersza. Drugą jest `isalnum`, testującą, czy znak jest alfanumeryczny, to znaczy czy jest cyfrą lub literą.

```
#include <stdio.h>
/* dla definicji isspace i isalnum */
#include <ctype.h>

#define MAXTOK      256

static char inbuf[MAXTOK+1];

char *getword(FILE *inf)
{
    int c, count = 0;
    /* wytnij odstęp */
    do {
        c = getc(inf);
    } while( isspace(c) );

    if(c == EOF)
        return (NULL);
    if( !isalnum(c) ) /* czy znak niealfanumeryczny */

```

```

inbuf[count++] = c;
else
{
    /* assemble "word" */
    do{
        if(count < MAXTOK)
            inbuf[count++] = c;

        c = getc(inf);

    } while( isalnum(c));
    ungetc(c, inf);      /* zwróć znak */

}

inbuf[count] = '\0';      /* zapewnij zwrot napisu */
return (inbuf);
}

```

Jeśli dostarczymy następujące wejście:

```

This is
the
input data!!!

```

procedura `getword` zwróci następującą sekwencję napisów:

```

This
is
the
input
data
!
!
!
```

Ćwiczenie 11.3 Zmodyfikuj procedurę `getword` tak, aby rozumiała liczby, które mogą zawierać początkowy znak minusa albo plusa oraz kropkę dziesiętną.

11.6 Standardowe wejście, standardowe wyjście i standardowe wyjście komunikatów o błędach

Standardowa Biblioteka I/O oferuje trzy struktury FILE połączone ze standardowym wejściem, standardowym wyjściem i standardowym wyjściem komunikatów o błędach. (Uważaj, abyś nie pogubił się w terminologii. Standardowa Biblioteka I/O i standardowe wejście to dwie całkowicie różne rzeczy). Te standardowe struktury FILE nie muszą być otwierane i są identyfikowane przez następujące wskaźniki FILE:

<code>stdin</code>	Odpowiada standardowemu wejściu.
<code>stdout</code>	Odpowiada standardowemu wyjściu.
<code>stderr</code>	Odpowiada standardowemu wyjściu komunikatów o błędach.

Poniższa instrukcja dostanie następny znak ze `stdin`, który, podobnie jak deskryptor pliku 0, domyślnie oznacza klawiaturę terminala:

```
inchar = getc(stdin);
```

Ponieważ `stdin` i `stdout` używa się bardzo często, dostarczane są dwie skrócone formy funkcji `getc` i `putc`, mianowicie `getchar` i `putchar`. Funkcja `getchar` zwraca następny znak ze `stdin`, a funkcja `putchar` umieszcza znak w `stdout`. Żadna z tych funkcji nie pobiera jako argumentu wskaźnika FILE.

Następujący program `io2` używa funkcji `getchar` i `putchar` do kopowania swojego standardowego wejścia do swojego standardowego wyjścia:

```

/* io2 -- kopiuje stdin do stdout */
#include <stdio.h>
main()
{
    int c;
    while( (c = getchar()) != EOF)
        putchar(c);
}

```

Po skompilowaniu program `io2` będzie zachowywał się mniej więcej jak `io`, wcześniejszy przykład z rozdziału 2.

Podobnie jak `getc` i `putc`, także `getchar` i `putchar` mogą być zdefiniowane jak makroinstrukcje. W rzeczywistości `getchar()` zwykle będzie rozwijane do `getc(stdin)`, a `putchar(c)` – podobnie – do `putc(c, stdout)`.

Wskaźnik `stderr` jest przeznaczony dla komunikatów o błędach. Z powodu tej specjalnej funkcji wyjście `stderr` zwykle nie jest buforowane. Innymi słowy, znak wysłany do `stderr` będzie natychmiast zapisany do pliku lub urządzenia aktualnie związanego ze standardowym wyjściem komunikatów o błędach. Jeśli lubisz wstawać w swoim kodzie dla celów testowych instrukcję tymczasowego śledzenia, wtedy wskazany jest zapis do `stderr`. Wyjście `stdout` jest buforowane i może być wyświetlone z opóźnieniem. (Ewentualnie można wykorzystać funkcję `fflush(stdout)` do opróżnienia bufora `stdout` ze wszystkich trzymanych w nim komunikatów).

Ćwiczenie 11.4 Używając standardowego polecenia `time` porównaj wydajność programu `io2` z programem `io` z rozdziału 2. Dostosuj oryginalną wersję programu `io` tak, aby używala funkcji `read` i `write` do odczytywania, a następnie zapisywania naraz pojedynczego znaku. Jak porównać wydajność tej wersji oraz programu `io2`?

Ćwiczenie 11.5 Zmodyfikuj program `io2` tak, aby dokładniej przypominał polecenie `cat`. W szczególności, aby drukował zawartość każdego pliku wymienionego w argumentach jego wiersza polecień. Jeśli nie są podane żadne argumenty, jego domyślnym wejściem powinno być `stdin`.

11.7 Standardowe procedury stanu I/O

Dostarczana jest pewna liczba prostych procedur, które pozwalają na określenie stanu struktury FILE. W szczególności pozwalają one programowi określić, czy procedura wejściowa Standardowego I/O jak `getc` zwróciła EOF, ponieważ osiągnęła koniec pliku, czy też dlatego, że wystąpił błąd. Dostępne procedury wymienione są poniżej:

Użycie

```
#include <stdio.h>
int ferror(FILE *stream);
int feof(FILE *stream);
void clearerr(FILE *stream);
int fileno(FILE *stream);
```

Funkcja boole'owska `ferror` zwraca wartość różną od zera (to znaczy *true*), jeśli na skutek poprzedniego żądania wejścia lub wyjścia w `stream` zdarzył się błąd.

Mógł on powstać, jeśli w procedurze Standardowego I/O zawiodło wywołanie pierwotnych funkcji dostępu do pliku (jak `read`, `write`, itp.). I odwrotnie, jeśli `ferror` zwraca zero (to znaczy *false*), oznacza to, że żaden błąd się nie zdarzył. Funkcja `ferror` może być używana następująco:

```
if(ferror(stream))
{
    /* obsługa błędu */
    ...
}
else
{
    /* gałąź bez błędu */
    ...
}
```

Funkcja boole'owska `feof` zwraca wartość różną od zera, jeśli poprzednio w `stream` został napotkany warunek oznaczający koniec pliku. Zwrot wartości zero wskazuje po prostu, że koniec pliku (EOF) nie został osiągnięty.

Funkcja `clearerr` jest używana do zerowania znacznika błędu i znacznika końca pliku w `stream`. Zapewnia to, że kolejne wywołania funkcji `ferror` i `feof` dla tego pliku będą zwracać 0, jeśli w międzyczasie nie zdarzy się inny wyjątek. Z przyczyn oczywistych `clearerr` nie jest często używaną procedurą.

Funkcja `fileno` nieco odstaje od pozostałych, ponieważ nie jest związana z obsługą błędów. Zwraca ona całkowity deskryptor pliku znajdujący się we wskazującej do `stream` strukturze FILE. Przydaje się to, jeśli musisz przekazać do procedury deskryptor pliku zamiast wskaźnika FILE. Jednak nie używaj `fileno` do mieszania pierwotnych funkcji dostępu do pliku i procedur Standardowego I/O. Niemal na pewno spowoduje to chaos.

Następujący przykład, `egetc`, używa funkcji `ferror` w celu rozróżnienia między błędem i prawdziwym końcem pliku w sytuacji, gdy procedura Standardowego I/O zwraca EOF:

```
/* egetc -- getc ze sprawdzaniem błędu */

#include <stdio.h>

int egetc(FILE *stream)
{
    int c;
    c = getc(stream);
    if(c == EOF)
    {
        if(ferror(stream))
        {
            fprintf(stderr, "fatal error: input error \n");
            exit(1);
        }
        else
            fprintf(stderr, "warning: EOF\n");
    }
    return (c);
}
```

Zwróć uwagę, że wszystkie funkcje opisywane w tym podrozdziale są zwykle implementowane jako makroinstrukcje i w związku z tym dotyczą ich wszystkie typowe ostrzeżenia.

11.8 Wejście i wyjście linii

Z procedurami I/O dotyczącymi pojedynczego znaku pokrewna jest pewna liczba prostych procedur dla odczytu i zapisu linii danych (linii będącej prostą sekwencją znaków, zakończoną znakiem nowego wiersza). Procedury te są odpowiednie dla programów interakcyjnych, które czytają z klawiatury i zapisują na ekran terminala. Podstawowe procedury wejścia linii to `gets` i `fgets`.

Użycie

```
#include <stdio.h>
char * gets(char *buf);
char * fgets(char *buf, int nsize, FILE *inf);
```

Funkcja `gets` odczytuje sekwencję znaków ze standardowego wejścia (`stdin`), umieszczając każdy znak w buforze wskazywanym przez `buf`. Znaki są czytane aż do napotkania znaku nowego wiersza lub końca pliku. Następnie znak `\n` zostaje usunięty, a w buforze `buf` umieszczany jest znak zerowy, dając prawidłowo sformatowany (czyli zakończony znakiem zerowym) napis. Jeśli wywołanie było pomyślne, zwróci wskaźnik do `buf`. Natomiast jeśli zdarzył się błąd lub został

napotkany koniec pliku i żadne znaki nie były odczytane, w zamian zwracany jest wskaźnik NULL.

Funkcja fgets jest uogólnioną wersją gets. Wczytuje ona znaki z inf do bufora buf, aż do odczytania nsize-1 znaków lub napotkania znaku newline albo końca pliku. Przy funkcji fgets znaki nowego wiersza nie są usuwane, tylko umieszczane na końcu bufora (pomaga to funkcji wywołującej określić warunek, który spowodował powrót funkcji fgets). Podobnie jak w przypadku gets, funkcja fgets zwraca wskaźnik do buf, jeśli wywołanie było pomyślne albo NULL w przeciwnym przypadku.

Funkcja gets należy raczej do pierwotnych. Ponieważ nie zna ona długości przekazywanego jej bufora, niespodziewanie dłuża linia może spowodować poważny błąd wewnętrzny. Dla bezpieczeństwa powinna więc być używana zamiast niej funkcja fgets (w połączeniu ze stdin).

Następująca procedura, yesno, używa w ten sposób fgets, aby uzyskać odpowiedź od użytkownika; wywołuje ona też isspace, aby pominąć odstępy (białe spacje) w odpowiedzi.

```
/* yesno -- pobiera odpowiedź 'yes' lub 'no' od użytkownika */
```

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

```
#define YES      1
#define NO       0
#define ANWSZ     80
```

```
static char *pdefault = "Type 'y' for YES, 'n' for NO";
static char *error = "Unexpected response";
```

```
int yesno(char *prompt)
{
    char buf[ANWSZ], *p_use, *p;
    /* jeśli prompt (znak zachęty) różny od NULL, użyj go.
     * W przeciwnym przypadku użyj pdefault */
    p_use = (prompt != NULL) ? prompt : pdefault;
    /* pętla aż do prawidłowej odpowiedzi */

    for(;;)
    {
        /* drukuj znak zachęty */
        printf("%s > ", p_use);
        if( fgets(buf, ANWSZ, stdin) == NULL )
            return EOF;
        /* wytnij początkowe odstępy */
        for(p = buf; isspace(*p); p++)
            ;
    }
}
```

```
        case 'Y':
        case 'y':
            return (YES);
        case 'N':
        case 'n':
            return (NO);
        default:
            printf("\n%s\n", error);
    }
}
```

Przykład zakłada, że standardowe wejście (stdin) jest połączone z terminaliem. Jak możesz wykonać to bezpieczniej?

Odwrotne procedury dla gets i fgets to odpowiednio puts i fputs.

Użycie

```
#include <stdio.h>
int puts(const char *string);
int fputs(const char *string, FILE *outf);
```

Funkcja puts zapisuje znaki napisu string do standardowego wyjścia (stdout), bez końcowego znaku zerowego. Natomiast funkcja fputs zapisuje string do pliku wskazywanego przez outf. Aby zapewnić zgodność ze starszymi wersjami systemu, puts dodaje na końcu znak nowego wiersza, podczas gdy fputs nie dodaje nic. Obie funkcje w przypadku błędu zwracają EOF.

Następujące wywołanie funkcji puts powoduje wydruk na standardowym wyjściu komunikatu Hello, world. Znak nowego wiersza jest dodawany automatycznie:

```
puts("Hello, world");
```

11.9 Wejście i wyjście binarne: fread i fwrite

Użycie

```
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t nitems,
             FILE *inf);
size_t fwrite(const void *buffer, size_t size, size_t nitems,
             FILE *outf);
```

Te dwie bardzo przydatne procedury umożliwiają binarne wejście i wyjście. Procedura fread odczytuje nitems obiektów danych z pliku wejściowego odpowiadają-

jęcego inf. Odczytywane bajty są umieszczane w tablicy buffer. Każdy odczytywany obiekt jest reprezentowany przez sekwencję bajtów o długości length. Wartość zwracana w result podaje liczbę pomyślnie odczytanych obiektów.

Procedura fwrite jest dokładną odwrotnością fread. Zapisuje ona dane zawarte w buffer do pliku wyjściowego wskazywanego przez outf. Bufor ten jest traktowany jako składający się z n items obiektów o długości size bajtów każdy. Zwieracana wartość podaje liczbę faktycznie zapisanych rekordów.

Typowo procedury te są używane do przekazywania zawartości dowolnych struktur danych w C do i z pliku binarnego. W takich okolicznościach parametr size jest zastępowany przez wywołanie sizeof, które podaje wielkość struktury w bajtach.

Następny przykład wskazuje, jak te procedury działają. Skupia się on na szablonie struktury dict_elem. Występowanie tej struktury może być używane do przedstawienia części rekordu w prostym systemie bazy danych. Używając terminologii bazy danych powiemy, że struktura dict_elem jest przeznaczona do opisu pola lub atrybutu bazy danych. Definicję dict_elem zamieściliśmy w pliku nagłówkowym o nazwie dict.h, który wygląda następująco:

```
/* dict.h -- nagłówek dla procedur słownika danych */

#include <stdio.h>

/* dict_elem -- element słownika danych */
/* opisuje pole w rekordzie bazy danych */

struct dict_elem{
    char d_name[15];      /* nazwa składowej słownika */
    int d_start;           /* początkowa pozycja w rekordzie */
    int d_length;          /* długość pola */
    int d_type;            /* typ danych */
};

#define ERROR      (-1)
#define SUCCESS     0
```

Bez wchodzenia w szczegóły budowy struktury wprowadzimy dwie procedury writedict i readdict, które odpowiednio zapisują i odczytują tablicę struktur dict_elem. Pliki tworzone przez te dwie procedury mogą stanowić prosty słownik danych dla rekordów w systemie bazy danych.

Procedura writedict pobiera dwa parametry, nazwę pliku wyjściowego i adres tablicy struktur dict_elem. Zakładamy, że ta lista kończy się na pierwszej strukturze w tablicy, w której składowa d_length jest równa zeru:

```
#include "dict.h"

int writedict(const char *dictname, struct dict_elem *elist)
{
    int j;
    FILE *outf;

    /* otwórz plik wyjściowy */
    if( (outf = fopen(dictname, "w")) == NULL)
```

```
        return ERROR;
    /* oblicz długość tablicy */
    for( j = 0; elist[j].d_length != 0; j++)
    ;

    /* zapisz listę struktur dict_elem */
    if( fwrite((void *)elist, sizeof(struct dict_elem), j, outf) < j)
    {
        fclose(outf);
        return ERROR;
    }

    fclose(outf);
    return SUCCESS;
}
```

Zwróć uwagę, że adres tablicy elist jest rzutowany do wskaźnika void za pomocą (void *). Użycie sizeof(struct dict_elem) służy do przekazania fwrite liczby bajtów zajmowanych przez strukturę dict_elem.

Procedura readdict używa fread do odtworzenia listy struktur z pliku. Pobiera ona trzy parametry: indictname, wskazujący nazwę pliku słownika, inlist, wskazujący tablicę struktur dict_elem, do której lista dyskowa będzie skierowana, i maxlen, wskazujący długość tej tablicy.

```
struct dict_elem *readdict(const char *indictname,
                           struct dict_elem *inlist,
                           int maxlen)

{
    int i;
    FILE *inf;

    /* otwórz plik wejściowy */
    if( (inf = fopen(indictname, "r")) == NULL)
        return NULL;

    /* wczytaj struktury dict_elem z pliku */
    for( i = 0; i < maxlen - 1; i++)
        if( fread((void *)&inlist[i], sizeof(struct dict_elem),
                  1, inf) < 1)
            break;

    fclose(inf);

    /* zaznacz koniec listy */
    inlist[i].d_length = 0;

    /* zwróć początek listy */
    return inlist;
}
```

I znów zwróć uwagę na użycie rzutowania i sizeof.

Powinniśmy dodać do naszego wywodu istotne zastrzeżenie. Dane binarne zapisywane do pliku za pomocą fwrite odzwierciedlają sposób pamiętania danych w pamięci systemu. Ponieważ ten sposób jest zależny od komputera (chodzi o kolej-

ność bajtów i wypełnianie), dane zapisane na jednym komputerze mogą nie być czytelne na drugim, dopóki nie zwróciśmy należytej uwagi na zapis informacji w formacie niezależnym od maszyny. Również adresy pamiętane we wskaźnikach nie powinny być z oczywistych powodów odczytywane i zapisywane.

Na zakończenie – jeszcze jedna uwaga: aby uzyskać podobny wynik, możemy użyć bezpośrednio funkcji `read` i `write`. Na przykład:

```
write(fd, (void *)ptr, sizeof(struct dict_elem));
```

Główną zaletą wersji Standardowego I/O jest – niezmiennie – efektywność. Ostatecznie dane będą odczytywane i zapisywane wielkimi blokami, niezależnie od wielkości struktury `dict_elem`.

Ćwiczenie 11.6 Obecne wersje `writedict` i `readdirct` manipuluują plikami słownika, które naprawdę opisują tylko jeden typ rekordu. Zmodyfikuj je tak, aby w tym samym pliku można było przechowywać informacje z kilku typów rekordów. Innymi słowy, pozwól plikowi słownika zawierać kilka niezależnych, nazwanych list struktur `dict_elem`. (Wskazówka: na początku pliku umieść strukturę „header” (nagłówek), zawierającą informację o liczbie rekordów i typie pól).

11.10 Bezpośredni dostęp do pliku: `fseek`, `rewind` i `ftell`

Standardowa Biblioteka I/O dostarcza procedur dla bezpośredniego dostępu, pozwalających programiście pozycjonować wskaźnik pliku w strumieniu pliku lub znajdować jego bieżącą pozycję. Procedurami tymi są: `fseek`, `rewind` i `ftell`. Mogą być one używane tylko z plikami, które obsługują bezpośredni dostęp (co na przykład wyklucza terminal).

Użycie

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int direction);
void rewind(FILE *stream); long ftell(FILE *stream);
```

Procedura `fseek` jest bliźniaczą procedurą dla swojego niskopoziomowego odpowiednika `lseek` i ustawia wskaźnik pliku w pliku powiązanym ze strumieniem. Dlatego przeddefiniuje położenie następnej operacji wejściowej lub wyjściowej. Parametr `direction` określa punkt startowy, od którego nowa pozycja w pliku ma być obliczona. Jeśli ma on wartość `SEEK_SET` (typowa wartość 0), punkt startowy stanowi początek pliku; jeśli jest to `SEEK_CUR` (typowa wartość 1), używana jest bieżąca pozycja; jeśli jest to `SEEK_END` (typowa wartość 2), punkt startowy stanowi koniec pliku. Przesunięcie `offset` podaje liczbę bajtów, które należy dodać do tego położenia. Podobnie jak przy `lseek`, ta wartość może być dowolną prawidłową liczbą typu `long int`, włącznie z wartościami ujemnymi. W normalnych okolicznościach `fseek` zwraca zero. Wartość niezerowa wskazuje błąd.

Procedura `rewind`(`stream`) jest prostym skrótem dla:

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

Innymi słowy, ustawia ona ponownie wskaźnik odczytu-zapisu na początek pliku.

Procedura `ftell` zwraca bieżącą logiczną pozycję programu w strumieniu pliku. Podaje ona liczbę bajtów od początku pliku, zaczynając liczenie od zera.

11.11 Formatowane wyjście: rodzina funkcji `printf`

Użycie

```
#include <stdio.h>
/* uwaga: parametry arg1 .. są arbitralnego typu */
int printf(const char *fmt, arg1, arg2 ... argn);
int fprintf(FILE *outf, const char *fmt, arg1, arg2 ... argn);
int sprintf(char *string, const char *fmt, arg1, arg2 ... argn);
```

Procedury te pobierają napis formatujący `fmt` i zmienną liczbę argumentów arbitralnego typu (wskaazywaną tu przez `arg1`, `arg2` itd.), aby utworzyć napis wyjściowy. Napis ten będzie przekazywał informację zawartą w parametrach `arg1` do `argn` używając formatu określonego w `fmt`. Za pomocą `printf` napis ten jest następnie kopiowany do `stdout`, a za pomocą `sprintf` – do pliku identyfikowanego przez `outf`. Jeśli chodzi o `sprintf`, to w rzeczywistości nie jest to w ogóle funkcja wyjściowa. W zamian napis „wyjściowy” tworzony przez `sprintf` będzie kopiowany do tablicy znaków wskaazywanej przez wskaźnik `string`. Dla wygody programowania `sprintf` dodaje też automatycznie końcowy znak zerowy.

Argument `fmt` stanowiący napis formatujący podobny jest w budowie do formatów, którymi dysponował język Fortran. Zawiera on mieszankę zwykłych znaków, które są kopowane dosłownie, i szeregu specyfikatorów konwersji (ang. *conversion specifications*). Są to podnapisy, zaczynające się zawsze od znaku procent % (jeśli chcesz wydrukować znak procentu, musisz przedstawić go za pomocą dwóch znaków procentu %%).

W napisie formatującym powinno znajdować się po jednym specyfikatorze konwersji dla każdego z argumentów `arg1`, `arg2` itd. Każdy z tych specyfikatorów określa typ odpowiadającego argumentu oraz sposób odwzorowania go na wyjściową sekwencję znaków ASCII.

Przed omówieniem ogólnej formy tych specyfikacji przedstawimy przykład demonstrujący użycie formatu `printf` dla dwóch prostych przypadków. W pierwszym nie ma żadnych argumentów poza samym napisem `fmt`. W drugim jest jeden dodatkowy argument: zmienna całkowita `iarg`.

```

int iarg = 34;
.

.

printf("Hello, world!\n");
printf("The variable iarg has the value %d\n", iarg);

```

Ponieważ w pierwszym wywołaniu nie występują żadne argumenty do konwersji, nie ma też żadnego specyfikatora konwersji w napisie formatującym. Dlatego instrukcja ta po prostu powoduje wyświetlenie na standardowym wyjściu komunikatu:

Hello, world!

zakończonego znakiem nowego wiersza (symbol `\n` w napisie jest interpretowany przez C jako oznaczający nowy wiersz). W drugiej instrukcji `printf` jest jeden dodatkowy argument `iarg` i dlatego w napisie formatującym znajduje się jeden specyfikator konwersji, mianowicie `%d`. Mówiąc procedurze `printf`, że dodatkowy argument jest zmienną całkowitą, która ma być drukowana w formie dziesiętnej (stąd użycie litery `d`). Dlatego wyjście tej instrukcji wygląda następująco:

The variable iarg has the value 34

Oto różne możliwe formy specyfikatora konwersji:

Konwersje całkowite

- `%d` Jak widzieliśmy, jest to standardowy kod konwersji dla liczby całkowitej ze znakiem (`int`). Jeśli wartość jest ujemna, automatycznie będzie dodany znak minus.
- `%u` Argument typu `unsigned int` ma być wydrukowany w formie dziesiętnej.
- `%o` Argument jest liczbą całkowitą i ma być wydrukowany w formie ósemkowej bez znaku.
- `%x` Argument jest liczbą całkowitą i ma być wydrukowany w formie szesnastkowej bez znaku. Dla dodatkowych cyfr szesnastkowych będą używane litery `a`, `b`, `c`, `d`, `e` i `f`. Jeśli w zamian podana zostanie specyfikacja `%X`, wtedy dodatkowymi cyframi szesnastkowymi będą `A`, `B`, `C`, `D`, `E` i `F`.
- `%ld` Argument jest liczbą całkowitą ze znakiem typu `long`, która będzie wydrukowana w formie dziesiętnej. Programista może też używać `%lo`, `%lu`, `%lx` i `%lx`.

Konwersje zmiennoprzecinkowe

- `%f` Argument jest typu `float` lub `double` i ma być wydrukowany w standarowej dziesiętnej formie zmiennoprzecinkowej.
- `%e` Argument jest typu `float` lub `double` i ma być wydrukowany w formie wykładniczej, przyjętej w zastosowaniach naukowych. Do oznaczenia potęgi będzie użыта litera `e`. Jeśli w zamian podany jest specyfikator `%E`, zamiast niej będzie użыта wielka litera `E`.

`%g` Jest to mieszanina specyfikacji `%e` i `%f`. Wskazuje, że argument jest zgodny z `float` lub `double`. W zależności od wielkości liczby będzie użyta notacja zmiennoprzecinkowa lub wykładnicza (jak dla `%e`). Jeśli w zamian podany jest specyfikator `%G`, w notacji wykładniczej będzie użyty styl `%E`.

Kontrola znaków i napisów

`%c` Argument jest typu `char` i ma być wydrukowany dokładnie taki, jak jest, nawet jeśli to znak niedrukowalny. Wartość liczbową zawartą w znaku może być wyświetlona za pomocą kodu konwersji całkowitej. Jest to przydatne, jeśli znak nie ma znaczącej reprezentacji na twoim terminalu.

`%s` Odpowiadający argument jest traktowany jako napis (to znaczy wskaźnik znakowy). Zawartość tego napisu będzie przekazywana dosłownie do wyjścia strumienia. Oczywiście napis musi być zakończony znakiem zerowym.

Następna przykładowa procedura, `warnuser`, pokazuje zastosowanie konwersji `%c` i `%s`. Używa ona `fprintf` do druku komunikatów ostrzegawczych na standardowym wyjściu komunikatów o błędach za pomocą pliku strumienia `stderr`. Jeśli `stderr` odpowiada terminalowi, procedura próbuje też dać trzykrotnie sygnał dźwiękowy za pomocą wysłania znaku `Ctrl+G` (ASCII `BEL`, który ma wartość `0x07` szesnastkowo). Procedura używa funkcji `isatty`, określającej, czy deskryptor pliku odpowiada terminalowi, i funkcji `fileno`, zwracającej deskryptor pliku związany ze strumieniem pliku. Funkcja `isatty` jest standardową funkcją Uniksa omawianą w rozdziale 9, podczas gdy `fileno` stanowi część samej Standardowej Biblioteki I/O, opisywaną w podrozdziale 11.7.

```
/* warnuser -- sygnalizuje dźwiękiem i drukuje komunikat */
```

```
#include <stdio.h>

/* to działa w większości terminali */
const char bel = 0x07;
void warnuser(const char *string)
{
    /* czy to jest terminal?? */
    if(isatty(fileno(stderr)))
        fprintf(stderr, "%c%c%c", bel, bel, bel);

    fprintf(stderr, "warning: %s\n", string);
}
```

Specyfikacja szerokości pola i precyzji

Specyfikatory konwersji mogą też zawierać informację o minimalnej szerokości w znakach pola, w którym argument jest drukowany i precyzji dla tego pola. W przypadku argumentu całkowitego precyzja oznacza minimalną liczbę występujących cyfr. W przypadku argumentu typu `float` lub `double`, precyzja podaje liczbę cyfr występujących po kropce dziesiętnej. Przy argumentach napisowych podaje maksymalną liczbę znaków, która może być pobrana z napisu.

W specyfikacji konwersji informacja o szerokości pola i precyzji występuje bezpośrednio po znaku procentu i jest rozdzielona kropką dziesiętną, na przykład:

%10.5d

oznacza: wydrukuj odpowiadający argument `int` w polu o szerokości 10 znaków; jeśli argument ma mniej niż pięć cyfr, wypełnij go wiodącymi zerami. Specyfikacja:

%5f

oznacza: wydrukuj odpowiadający argument `float` lub `argument double` z pięcioma miejscami dziesiętnymi. Ten konkretny przykład pokazuje, że część dotycząca precyzji może występuwać samodzielnie. Podobnie może być podana tylko szerokość pola. Specyfikacja:

%10s

oznacza więc: wydrukuj odpowiadający napis w polu o minimalnej szerokości 10 znaków.

Wszystkie powyższe przykłady będą tworzyć wyjście, które jest wyrównane do prawej strony wyspecyfikowanego pola znakowego. Aby zapewnić wyrównanie do lewej, bezpośrednio po znaku procentu musi wystąpić znak minus. Tak więc specyfikacja konwersji:

%-30s

oznacza, że odpowiadający argument napisowy będzie wydrukowany po lewej stronie pola o szerokości co najmniej 30 znaków.

Czasami część specyfikacji konwersji dotycząca szerokości pola nie może być obliczona przed uruchomieniem programu. Aby to obejść, specyfikacja szerokości może być zastąpiona gwiazdką (*). Wtedy `printf` będzie oczekiwany, że gwiazdka jest zgodna ze zmiennej całkowitą dającą właściwą szerokość pola. Tak więc:

int width, iarg;

```
.
.
.

.printf("%*d", width, iarg);
```

powoduje, że liczba całkowita `iarg` będzie wydrukowana w polu o szerokości `width` znaków.

Przykład pakietu oszczędnościowego

Liczba permutacji różnych formatów jest oczywiście ogromna, więc aby zaoszczędzić miejsce, upchnęliśmy szereg przykładów w następnym, całkowicie nirealistycznym przykładowym programie. Funkcja `atan` to standardowa funkcja arcus tangens z biblioteki matematycznej Uniksa. (Z tego powodu, jeśli spróbujesz skompilować ten przykład, będziesz musiał dodać do wywołania `cc` opcję `-lm` lub zbliżoną, aby w trakcie łączenia została załadowana biblioteka matematyczna).

```
/* cram -- pakiet oszczędnościowy demonstrujący printf */

#include <stdio.h>
#include <math.h>

main()
{
    char *weekday = "Sunday";
    char *month = "September";
```

```
char *string = "Hello, world";

int i = 11058;
int day = 15, hour = 16, minute = 25;

/* drukuj datę */
printf("Date is %s, %d %s, %d:%.2d\n",
       weekday, day, month, hour, minute);

/* drukuj znowu znak nowego wiersza */
putchar('\n');

/* pokaż oddziaływanie szerokości i precyzji na napis */
printf(">>%s<<\n", string);
printf(">>%30s<<\n", string);
printf(">>%-30s<<\n", string);
printf(">>%30.5s<<\n", string);
printf(">>%-30.5s<<\n", string);

putchar('\n');
/* wydrukuj i w różnych formach */
printf("%d, %u, %o, %x, %X\n", i, i, i, i, i);

/* wydrukuj pi do 5 miejsc dziesiętnych */
printf("PI is %.5f\n", 4*atan(1.0));
}
```

Program ten daje następujący wydruk:

Date is Sunday, 15 September, 16:25

```
>>Hello, world<<
>>                                Hello, world<<
>>Hello, world                         <<
>>                                Hello<<
>>Hello                               <<
11058, 11058, 25462, 2b32, 2B32
PI is 3.14159
```

Symboli specjalne

Specyfikacje konwersji wyjściowej mogą być jeszcze bardziej komplikowane przez kilka dodatkowych symboli specjalnych. Jednym z nich jest znak hash, czyli `#`. Musi on wystąpić bezpośrednio przed częścią specyfikacji dotyczącej szerokości pola. Dla liczb całkowitych bez znaku przy kodach konwersji `o`, `x` i `X` powoduje on automatyczne poprzedzenie wyjścia odpowiednio znakami `0`, `0x` lub `0X`. Tak więc fragment kodu:

```
int arg = 0xFF;
printf("In octal, %#o\n", arg);
```

daje następujące wyjście:

In octal, 0377

Przy konwersji zmiennoprzecinkowej znak `#` wymusza wydruk kropki dziesiętnej, nawet przy zerowej precyzji.

W specyfikacji konwersji można także wprowadzić znak plus (+), aby wymusić druk znaku + przed liczbami dodatnimi. (Ma to znaczenie tylko przy zmiennych całkowitych ze znakiem lub liczbach zmiennoprzecinkowych). Zajmuje on raczej szczególne miejsce w specyfikacji konwersji, występując bezpośrednio za znakiem minus wskazującym wyrównanie do lewej lub symbolem procentu, jeśli znak minus jest nieobecny. Następujące linie kodu:

```
float farg = 57.88;
printf("Value of farg is %+.10.2f\n", farg);
```

dadzą w wyniku:

```
Value of farg is +57.88
```

Zwróć uwagę na kombinację symboli minus i plus. Symbol + może być także zastąpiony za pomocą spacji. W tym przypadku procedura printf będzie drukować spację w miejscu, gdzie powinien wystąpić znak plus. Pozwala to na prawidłowe wyrównanie liczb dodatnich i ujemnych.

Procedura sprintf

Teraz omówimy krótko procedurę sprintf. Nie myśl o niej jako o procedurze wyjściowej. W rzeczywistości dostarcza ona najbardziej elastycznej manipulacji napisami i ogólnej konwersji ze wszystkich bibliotek C. Następujący fragment kodu wskazuje, jak można jej użyć:

```
/* genkey -- generuje klucz do użycia w bazie danych */
/*           klucz ma zawsze 20 znaków długości */

#include <stdio.h>
#include <string.h>
char * genkey(char *buf, const char *suppcode, long orderno)
{
    /* czy suppcode jest prawidłowy? */
    if(strlen(suppcode) != 10)
        return (NULL);

    sprintf(buf, "%s.%ld", suppcode, orderno);

    return (buf);
}
```

Następujące wywołanie genkey:

```
printf("%s\n", genkey(buf, "abcdefgij", 12));
```

spowoduje wyświetlenie następującego napisu:

```
abcdefgij_000000012
```

11.12 Formatowane wejście: rodzina funkcji scanf

Użycie

```
#include <stdio.h>

/* Uwaga: ptr1 .. ptrn są wskaźnikami.
 * Typ wskazywanej przez nie zmiennej
 * jest arbitralny.
 */
int scanf(const char *fmt, ptr1, ptr2, ... ptrn);
int fscanf(FILE *inf, const char *fmt, ptr1, ptr2 ... ptrn);
int sscanf(const char *string, const char *fmt, ptr1, ptr2 ... ptrn);
```

Procedury te są odwrotnością procedur w rodzinie printf. Wszystkie przyjmują wejście z pliku (lub napis w przypadku sscanf), dekodują zgodnie z informacją o formacie zawartą w napisie fmt i umieszczają wynik w zmiennych wskazywanych przez wskaźniki ptr1 do ptrn. Wskaźnik pliku powiększany jest o liczbę przetwarzanych znaków.

Procedura scanf zawsze czyta ze stdin. Procedura fscanf czyta z pliku wskazywanego przez inf. Procedura sscanf jest czarną owcą rodziny i dekoduje napis wskazywany przez string, a nie wejście z pliku. Ponieważ działa ona na napisie zawartym w pamięci, jest szczególnie przydatna, gdy napis wejściowy musi być czytany wielokrotnie.

Napis formatujący fmt jest podobny w strukturze do napisu formatującego używanego przez procedurę printf. Na przykład następująca instrukcja odczytuje ze standardowego wejścia kolejną liczbę całkowitą:

```
int inarg;
scanf("%d", &inarg);
```

Najważniejsze, na co należy tu zwrócić uwagę, to przekazywanie do scanf adresu inarg. Jest tak, ponieważ język C może przekazywać parametry tylko przez wartość, a nigdy przez referencję. Dlatego jeśli chcemy, aby procedura scanf zmieniła wartość zmiennej występującej w procedurze wywołującej, musimy przekazać jej wskaźnik, który zawiera adres tej zmiennej. Bardzo łatwo zapomnieć o znaku ampersand, co może spowodować defekt pamięci. Pełni zap Şu nowicjusze powinni oprzeć się też pokusie umieszczania znaku ampersand przed istniejącymi wskaźnikami albo adresami takimi jak nazwy tablic znaków.

Zasadniczo napis formatujący procedury scanf może zawierać:

1. **Znaki odstępu (białej spacji);** to znaczy spacje, tabulacje, znaki nowego wiersza i wysunięcia strony. Zwykle jest to zgodne ze wszystkimi odstępami począwszy od bieżącej pozycji w strumieniu wejściowym aż do pierwszego znaku nie będącego odstępem.

2. *Zwykłe znaki, nie będące odstępami.* Muszą one dokładnie odpowiadać znakom strumienia wejściowego.
3. *Specyfikatory konwersji.* Jak wspomnieliśmy wcześniej, są one bardzo podobne do specyfikatorów używanych w procedurze printf.

Następny przykład pokazuje użycie procedury scanf z kilkoma zmiennymi różnych typów:

```
/* przykładowy program dla scanf */
#include <stdio.h>

main()
{
    int i1, i2;
    float f1t;
    ...
    char str1[10], str2[10];
    scanf("%2d %2d %f %s %s", &i1, &i2, &f1t, str1, str2);
    ...
}
```

Pierwsze dwa specyfikatory konwersji w napisie formatującym każdą procedurze scanf szukać dwóch liczb całkowitych (w formacie dziesiętnym). Ponieważ w każdym przypadku podana jest szerokość pola równa dwóm znakom, zakłada się, że pierwsza liczba całkowita znajduje się na następnych dwóch odczytywanych pozycjach, a druga na następnych dwóch pozycjach po niej (zasadniczo szerokość pola wskazuje maksymalną liczbę znaków, którą wartość może zajmować). Specyfikacja %f oznacza zmienną typu float, natomiast %s wskazuje, że jest oczekiwany napis, ograniczony przez znaki odstępu. Jeśli więc ten program otrzyma następującą sekwencję wejściową:

11 12 34.07

keith ben

uzyskamy następujący wynik:

```
zmienna i1 będzie ustawiona na 11
zmienna i2 będzie ustawiona na 12
zmienna f1t będzie ustawiona na 34.07
napis str1 będzie zawierał "keith"
napis str2 będzie zawierał "ben"
```

Oba napisy są zakończone znakiem zerowym. Zwróć uwagę, że str1 i str2 muszą być zdefiniowane wystarczająco duże, aby zatrzymać oczekiwany napis i końcowy znak zerowy. Nie wystarczy przekazanie do sprintf niezainicjowanego wskaźnika znakowego.

Przy specyfikatorze konwersji %s oczekiwany jest napis zakończony znakiem odstępu. Aby wczytać odstępy i inne znaki, należy użyć kodu konwersji %c. Na przykład instrukcja:

```
scanf ("%10c", s1);
```

odczyta następnych 10 dowolnych znaków ze strumienia wejściowego i umieści je w tablicy znaków s1. Ponieważ kod konwersji c jest zgodny z odstępami, aby otrzymać następny znak nie będący odstępem, należy użyć specyfikatora %1s, na przykład:

```
/* odczytuje 2 znaki, poczynając od
   pierwszego znaku nie będącego odstępem */
scanf ("%1s%1c", &c1, &c2);
```

Inny sposób określania danych napisowych (jedyny, który nie ma odpowiednika w napisach formatujących używanych przy printf), to wzorzec przeglądania (ang. *scanf-set*). Jest on sekwencją znaków zawartą w nawiasach kwadratowych: [i]. W tym przypadku brane jest pole wejściowe zawierające najdłuższą sekwencję znaków dopasowaną do wzorca przeglądania (jeśli znaki odstępu są częścią wzorca, to nie są pomijane). Na przykład instrukcja:

```
scanf ("%[ab12]s", str1, str2);
```

przy następującym napisie wejściowym:

2bbaalother

umieści 2bbaal w napisie str1 i other w napisie str2.

Istnieje jeszcze kilka przydatnych konwencji (znanych użytkownikom programów grep i sed), używanych przy budowie wzorca przeglądania. Na przykład zakres znaków jest oznaczany za pomocą podnapisu w stylu *pierwszy-ostatni*. Tak więc wzorzec [a-d] jest równoważny wzorcowi [abcd]. Jeśli sam myślnik (-) ma być częścią wzorca, musi stanowić jego pierwszy lub ostatni znak. Podobnie, jeśli we wzorcu ma wystąpić nawias kwadratowy zamkający], musi on być pierwszym znakiem po nawiasie otwierającym [. Jeśli jako pierwszy we wzorcu występuje znak cyrkumfleks (^), wtedy wzorzec przeglądania zostaje przeddefiniowany i oznacza wszystkie znaki *nie występujące* w obecnym wzorcu przeglądania.

W celu przypisania do zmiennych całkowitych typu long i zmiennoprzecinkowych typu double, po symbolu procentu w odpowiedniej specyfikacji konwersji musi występować litera l. Umożliwia to procedurze scanf określenie wielkości parametru, z którym ma do czynienia. Następujący fragment programu pokazuje, jak odczytać zmienne obu typów ze strumienia wejściowego:

```
long l;
double d;
```

```
scanf ("%ld %lf", &l, &d);
```

Jeszcze inna często występująca sytuacja to taka, gdy strumień wejściowy zawiera więcej danych, niż interesuje programistę. W tym przypadku specyfikacja konwersji może zawierać bezpośrednio po początkowym symbolu procentu gwiazdkę (*). Wskazuje ona przypisanie eliminacji. W efekcie pole wejściowe zgodne ze specyfikacją jest ignorowane. Wywołanie:

```
scanf ("%d %*s %*d %s", &ivar, string);
```

w połączeniu z linią wejściową:

131 cat 132 mat

spowoduje, że `scanf` przypisze 131 do zmiennej `i.var`, opuści kolejne dwa pola, a następnie umieści `mat` w zmiennej `string`.

No a co z wartością zwracaną przez członków rodziny `scanf`? Zwykle będą zwracać liczbę pomyślnie zgodnych i przypisanych elementów. Wartość zwracana może być równa zeru, jeśli wystąpi wczesny konflikt między napisem formującym i rzeczywistym wejściem. Jeśli wejście zakończy się przed pomyślnym dopasowaniem konfliktowej konwersji, zwracany jest EOF.

Ćwiczenie 11.7 Napisz program, który pobiera swoje argumenty (powinny nimi być dziesiętne liczby całkowite) i wyświetla je w formacie szesnastkowym oraz ósemkowym.

Ćwiczenie 11.8 Napisz program `savematrix`, który powinien zapamiętać macierz liczb całkowitych o arbitralnej wielkości w pliku w czystym formacie, oraz program `readmatrix`, odzyskujący macierz z pliku. Do wykonania całej pracy wykorzystaj tylko procedury `fprintf` i `fscanf`. Spraw, żeby ilość odstępów (spacji, tabulacji, itd.) w pliku była minimalna. Wskazówka: przy zapisie pliku użyj symbolu zmiennej szerokości (*).

11.13 Uruchamianie programów za pomocą Standardowej Biblioteki I/O

Standardowa Biblioteka I/O dostarcza kilku procedur przeznaczonych do uruchamiania jednego programu z innego. Najbardziej podstawową z nich jest procedura używana już przez nas wcześniej: `system`, którą możemy traktować jako procedurę ogólnego zastosowania.

Użycie

```
#include <stdlib.h>
int system(const char *comstring);
```

Procedura `system` uruchamia polecenie zawarte w `comstring`. Wykonuje to za pomocą utworzenia najpierw procesu potomnego. Następnie wywołuje funkcję `exec`, aby uruchomić standardową powłokę Uniksa z parametrem `comstring` jako wejściem. Procedura `system` w pierwszym procesie wywołuje `wait`, aby zapewnić, że będzie kontynuowała wykonywanie po zakończeniu polecenia. Ewentualna wartość zwracana `retval` zawiera stan wyjścia z powłoki, na podstawie którego można określić, czy polecenie zostało wykonane pomyślnie. Jeśli wywołanie `fork` lub `exec` zawiedzie, `retval` powinno zawierać wartość -1.

Ponieważ jako proces pośredniczący wywoływana jest powłoka, `comstring` może być dowolnym poleceniem, które można wpisać z terminala. Umożliwia to progiście wykorzystanie takich ułatwień, jak rozszerzone nazwy plików,

przekierowanie I/O itp. Następująca instrukcja używa procedury `system` do utworzenia podkatalogu za pomocą programu `mkdir`:

```
if( (retval = system("mkdir workdir")) != 0)
    fprintf(stderr, "system returned %d\n", retval);
```

A teraz trochę szczegółów. Po pierwsze, procedura `system` w procesie *wywolującym* ignoruje sygnały `SIGINT` i `SIGQUIT`. Pozwala to użytkownikowi bezpiecznie przerwać polecenie bez niepokojenia procesu rodzicielskiego. Po drugie, polecenia uruchamiane przez procedurę `system` dziedziczą niektóre otwarte deskryptory pliku z procesu wywołującego. W szczególności polecenie będzie pobierać swoje standardowe wejście z tego samego źródła, co proces rodzicielski. Jeśli standardowym wejściem jest plik, może to stanowić problem w przypadku, kiedy procedury `system` używamy do uruchomienia programu interakcyjnego, ponieważ on też będzie pobierało swoje wejście z pliku. Następujący fragment programu pokazuje jedno z możliwych rozwiązań tego problemu. Jest tu użyta funkcja `fcntl` w celu zapewnienia, że standardowe wejście identyfikowane przez deskryptor pliku 0 odpowiada terminalowi sterującemu procesu `/dev/tty`. Wywołania przykładowej funkcji `fatal` omawialiśmy wcześniej.

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

.

.

int newfd, oldfd;

.

.

/* powiel bieżący deskryptor pliku dla standardowego wejścia */
if( (oldfd = fcntl(0, F_DUPFD, 0)) == -1)
    fatal("fcntl failed");

/* otwórz terminal sterujący */
if( (newfd = open("/dev/tty", O_RDONLY)) == -1)
    fatal("open failed");

/* zamknij standardowe wejście */
close(0);

/* utwórz nowe standardowe wejście */
if( (fcntl(newfd, F_DUPFD, 0) != 0)
    fatal("fcntl problem");

close(newfd);

/* uruchom edytor interakcyjny */
if(system("ed newfile") == -1)
    fatal("system failed");
/* przywróć poprzednie standardowe wejście */
close(0);
if(fcntl(oldfd, F_DUPFD, 0) != 0)
    fatal("fcntl problem");
close(oldfd);

.
```

edura system ma jedną wielką wadę. Nie pozwala ona programowi na bezpośredni dostęp do wyjścia generowanego przez uruchomione polecenie. Aby uzyskać ten dostęp, programista powinien użyć jeszcze dwóch procedur z standardowej Biblioteki I/O: popen i pclose.

Użycie

```
#include <stdio.h>
FILE *popen(const char *comstring, const char *type);
void pclose(FILE *strm);
```

Obecnie jak system, procedura popen tworzy proces połomny w celu uruchomienia polecenia wskazywanego przez comstring. Jednak w odróżnieniu od procedury system tworzy ona także potok między procesem wywołującym poleceniem. Zwraca wtedy strukturę FILE związaną z potokiem. Jeśli parametr type jest równy "w", program może zapisywać do standardowego wejścia polecenia przez strukturę FILE. Natomiast jeśli parametr type jest równy "r", program może odczytywać ze standardowego wyjścia polecenia. W ten sposób procedura popen dostarcza prostą, czystą metodę komunikacji z innym programem.

Zamknięcie strumienia, który był otwarty przez procedurę popen, zawsze powinno być używana procedura pclose. Powinna ona czekać na zakończenie polecenia, a następnie zwrócić jego stan wyjściowy. Następująca procedura przykładowa, getlist, używa procedury popen i poleceń ls do otrzymania wydruku katalogu. Każda nazwa pliku jest wtedy zapisywana w dwuwymiarowej tablicy znaków, której adres był przekazany do getlist jako parametr.

getlist -- procedura uzyskiwania nazw plików w katalogu */

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

#define MAXLEN 255 /* maksymalna długość nazwy pliku */
#define MAXCMD 100 /* maksymalna długość polecenia */
#define ERROR (-1)
#define SUCCESS 0
```

```
getlist(char *namepart, char dirnames[][MAXLEN+1],
        int maxnames)
```

```
char cmd[MAXCMD+1], inline[MAXLEN+2];
int i;
FILE *lsf;
/* pierwsza forma polecenia */
strcpy(cmd, "ls ");
/* dodaj dodatkową część polecenia */
```

```
if(namepart != NULL)
    strncat(cmd, namepart, MAXCMD - strlen(cmd));
if(( lsf = popen(cmd, "r")) == NULL) /* uruchom polecenie */
    return (ERROR);

for(i = 0; i < maxnames; i++)
{
    if(fgets(inline, MAXLEN+2, lsf) == NULL)
        break;

    /* usuń znaki nowego wiersza */
    if(inline[strlen(inline)-1] == '\n')
        inline[strlen(inline)-1] = '\0';

    strcpy(&dirnames[i][0], inline);
}

if(i < maxnames)
    dirnames[i][0] = '\0';

pclose(lsf);
return(SUCCESS);
}
```

Procedura getlist może być użyta w takim wywołaniu, jak:

```
getlist("*.c", namebuf, 100);
```

Umieści wtedy nazwy programów w C z bieżącego katalogu roboczego w namebuf. Następny przykład rozwiązuje wspólny problem administratorów systemu Unix: jak szybko odmrozić terminal, który został zablokowany, na przykład przez niewypróbowany program graficzny. Procedura unfreeze pobiera jako argumenty nazwę terminala i listę programów. Następnie uruchamia za pomocą procedury popen polecenie stanu procesu ps w celu uzyskania listy procesów związanych z terminaliem. Wyszukuje na tej liście procesy uruchomione przez ten program. Dla każdego z procesów, które spełniają to kryterium, procedura unfreeze pyta użytkownika, czy ten proces nie powinien zostać zakończony.

Polecenie ps jest bardzo zależne od systemu. Dzieje się tak dlatego, że bada ono bezpośrednio jądro (za pośrednictwem specjalnego pliku, który reprezentuje pamięć), aby uzyskać tablicę procesów systemu. W komputerze, którego używaliśmy, polecenie ps wyświetlające procesy związane z konkretnym terminaliem ma ogólną formę:

\$ ps -t ttyname

gdzie ttyname jest nazwą pliku specjalnego terminala z katalogu /dev, takiego jak tty01, console, pts/8 itp. W komputerze, na którym testowaliśmy przykład, ten rodzaj polecenia ps dał następujące wyniki:

PID	TTY	TIME	COMMAND
29	co	0:04	sh
39	co	0:49	vi
42	co	0:00	sh
43	co	0:01	ps

Kolumna 1 zawiera process-id (identyfikator procesu). Kolumna 2 zawiera nazwą terminala o którym mowa; tutaj co oznacza console. Kolumna 3 podaje łączny czas działania procesu, a kolumna 4 – nazwę działającego programu. Zwróć uwagę na pierwszą linię, stanowiącą nagłówek kolumn. Nie wystąpi ona w programie `unfreeze`, którego źródło wygląda następująco:

```
/* unfreeze -- odmraża terminal */

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

#define LINESZ 50
#define SUCCESS 0
#define ERROR (-1)

main(int argc, char **argv)
{
    /* zależna od systemu inicjacja */
    static char *pspart = "ps -t ";
    static char *fmt = "%d %s %s %s";

    char comline[LINESZ], inbuf[LINESZ], header[LINESZ];
    char name[LINESZ];
    FILE *f;
    int killflag = 0, j;
    pid_t pid;

    if(argc <= 2)
    {
        fprintf(stderr, "usage: %s tty program ...\n", argv[0]);
        exit(1);
    }

    /* zbuduj wiersz poleceń */
    strcpy(comline, pspart);
    strcat(comline, argv[1]);

    /* uruchom polecenie ps */
    if((f = popen(comline, "r")) == NULL)
    {
        fprintf(stderr, "%s: could not run ps program\n", argv[0]);
        exit(2);
    }

    /* pobierz i zignoruj pierwszą linię z ps */
    if(fgets(header, LINESZ, f) == NULL)
    {
        sprintf(stderr, "%s: no output from ps?\n", argv[0]);
        exit(3);
    }

    /* szukaj programu do usunięcia */
    while(fgets(inbuf, LINESZ, f) != NULL)
```

```
        if(sscanf(inbuf, fmt, &pid, name) < 2)
            break;

        for(j = 2; j < argc; j++)
        {
            if(strcmp(name, argv[j]) == 0)
            {
                if(dokill(pid, inbuf, header) == SUCCESS)
                    killflag++;
            }
        }
    }

    /* to jest ostrzeżenie, a nie błąd */
    if(!killflag)
        fprintf(stderr, "%s: no program killed on %s\n", argv[0],
                argv[1]);

    exit(0);
}
```

Wywoływana przez `unfreeze` procedura `dokill` jest zaimplementowana w pokazany poniżej sposób. Zwróć uwagę na użycie `scanf` do odczytania pierwszego znaku nie będącego odstępem (możemy w zamian użyć opisanej w podrozdziale 11.8 funkcji `yesno`).

```
/* potwierdza, a następnie usuwa */
int dokill(pid_t procid, const char *line, const char *hd)
{
    char c;
    printf("\nProcess running named program found :\n");
    printf("\t%s\t%s\n", hd, line);
    printf("Type 'y' to kill process %d\n", procid);
    printf("\nAnswer > ");
    /* pobierz następny znak nie-odstępu */
    scanf("%ls", c);

    if(c == 'y' || c == 'Y')
    {
        kill(procid, SIGKILL);
        return(SUCCESS);
    }
    return(ERROR);
}
```

Ćwiczenie 11.9 Napisz swoją własną wersję procedury `getcwd`, która zwraca napis zawierający nazwę bieżącego katalogu roboczego. Wywołaj swoją wersję `wdir`. Wskazówka: użyj standardowego polecenia `pwd`.

Ćwiczenie 11.10 Napisz program o nazwie `arrived`, który używa programu `who` w połączeniu z `popen` do sprawdzania (co 60 sekund), czy zarejestrowała się jedna albo więcej osób z listy użytkowników. Lista nazwisk powinna być przekazana do `arrived` przez argumenty wiersza poleceń. Kiedy program dostrzeże użytkownika

nika z listy, powinien wyświetlać komunikat. Twój program musi być efektywny. Upewnij się, że używasz sleep do zawieszenia wykonania między sprawdzaniem. Polecenie who jest opisane w podręczniku twojego systemu.

11.14 Rozmaite funkcje

W tym podrozdziale krótko opiszemy rozmaite funkcje Standardowej Biblioteki I/O. Więcej szczegółów znajdziesz w oficjalnej dokumentacji swojego systemu.

11.14.1 Funkcje freopen i fopen

Użycie

```
#include <stdio.h>
FILE * freopen(const char *filename, const char *type,
               FILE *oldstream);
FILE * fdopen(int filedes, const char *type);
```

Funkcja freopen zamknie oldstream, a następnie otwiera go ponownie dla wejścia z filename. Parametr type określa tryb dostępu dla nowej struktury FILE i przybiera te same wartości, co dla fopen (r, w itd.). Zwykle jest używana dla zmiany przydziału stdin, stdout albo stderr, na przykład:

```
if (freopen("new.input", "r", stdin) == NULL)
    fatal("stdin could not be reassigned");
```

Funkcja fdopen wiąże nową strukturę FILE z całkowitym deskryptorem pliku filedes, który był otrzymany z uprzedniego wywołania jednej z funkcji systemowych: creat, open, pipe albo dup2.

Obie procedury w przypadku błędu zwracają NULL.

11.14.2 Kontrola bufora: setbuf i setvbuf

Użycie

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf1);
int setvbuf(FILE *stream, char *buf2, int type, size_t size);
```

Obydwie te procedury pozwalają programistom kontrolować w pewnym stopniu buforowanie związane z plikiem. Muszą być używane po otwarciu pliku, ale przed odczytem lub zapisem do niego.

Procedura setbuf jest używana do zastąpienia przez buf1 bufora zwykle alokowanego przez Standardową Bibliotekę I/O. Wymaganą wielkość buf1 określa stała BUFSIZ w <stdio.h>.

Jeśli natomiast do setbuf jest przekazywany w zamian wskaźnik znakowy NULL, wtedy wejście albo wyjście będzie niebuforowane. Może to być przydatne podczas testowania programu, gdy program kończy się nietypowo i dane zawarte w buforze są tracone.

Procedura setvbuf pozwala na bardziej precyzyjną kontrolę niż setbuf. Parametr buf2 podaje adres opcjonalnego nowego bufora. Parametr size określa wielkość buf2. Jeśli zamiast rzeczywistego adresu zostanie przekazany wskaźnik NULL, będzie użyte buforowanie domyślne. Parametr type określa, jak jest buforowany stream. Może być używany w celu dopasowania strumienia pliku do plików dyskowych lub urządzeń terminala. Trzy dozwolone wartości tego parametru zostały określone w <stdio.h>. Są to:

IOWBF Strumień pliku będzie w pełni buforowany. Jest to domyślne dla wszystkich strumieni pliku nie powiązanych z terminaliem. Dane powinny być zapisywane lub odczytywane porcjami po BUFSIZ bajtów, aby zmaksymalizować efektywność.

IOLBF Wyjście będzie buforowane liniami, a bufor będzie opróżniany, ilekroć zostanie zapisany znak nowego wiersza. Będzie także opróżniany po zapelnieniu i kiedy nastąpi żądanie odczytu wejścia. Jest to domyślne dla terminala i zaprojektowane do pomocy w działaniu interakcyjnym.

IOBNF Powoduje niebuforowane wejście i wyjście. W tym przypadku parametry buf2 i size będą ignorowane. Jest to tryb odpowiedni między innymi do rejestracji błędów.

Zwrócić uwagę, że jeśli podana zostanie nielegalna wartość dla parametru type albo size, procedura setvbuf zwraca wartość niezerową. I odwrotnie, zero wskazuje tu powodzenie.

ROZDZIAŁ 12

Rozmaite funkcje systemowe i procedury biblioteczne

- 12.1 Wprowadzenie
- 12.2 Dynamiczne zarządzanie pamięcią
- 12.3 I/O odwzorowane w pamięci i manipulacja pamięcią
- 12.4 Czas
- 12.5 Napisy i manipulacja znakami
- 12.6 Inne przydatne funkcje

12.1 Wprowadzenie

W końcowym rozdziale omawiamy przydatne funkcje systemowe i procedury biblioteczne, które nie pasowały logicznie do poprzednich rozdziałów. Poruszone tematy zawierają zarządzanie pamięcią, funkcje czasu, zatwierdzanie znaków i manipulację napisami.

12.2 Dynamiczne zarządzanie pamięcią

Każdy z programów, które dotychczas omawialiśmy, używał struktur danych, w pełni określonych przez standardowe deklaracje języka C, takie jak:

```
struct something x, y, *z, a[20];
```

Innymi słowy, rozmieszczenie używanych przez nasze przykłady danych było określane w czasie komplikacji. Jednak wiele problemów obliczeniowych najlepiej rozwiązywać za pomocą dynamicznego tworzenia i usuwania struktur danych, co oznacza, że rozmieszczenie danych używanych przez program jest określone ostatecznie w czasie jego wykonywania. W systemie Unix rodzina funkcji bibliotecznych `malloc` – gdzie nazwa `malloc` oznacza przydział pamięci (ang. *memory allocation*) – pozwala programistom C tworzyć obiekty dynamicznie na stercie (ang. *heap*) programu. Sama funkcja `malloc` jest zdefiniowana następująco:

Użycie

```
#include <stdlib.h>
void *malloc(size_t nbytes);
```

Wywołanie powoduje zwykle to, że `malloc` zwraca wskaźnik do `nbytes` bajtów nowo przydzielonego ciąglego obszaru pamięci. W efekcie program może wykorzystać dodatkową tablicę znaków. Jeśli nie ma dostępnej wystarczającej ilości pamięci i `malloc` nie może przydzielić żądaną ilość, zwraca w zamian pusty wskaźnik (`NULL`).

Funkcja `malloc` jest jednak o wiele częściej wykorzystywana do utworzenia obszaru pamięci zawierającego jedną (albo więcej) struktur danych, na przykład:

```
struct item *p;
p = (struct item *) malloc(sizeof(struct item));
```

Pomyślne wywołanie `malloc` tworzy nową strukturę `item`, która może być referowana za pomocą wskaźnika `p`. Zwróć uwagę, że wartość zwracana przez `malloc` jest rzutowana na wskaźnik odpowiedniego typu. Pozwala to uniknąć ostrzeżeń kompilatora albo takich narzędzi jak `lint`. Rzutowanie jest ważne, ponieważ funkcja `malloc` została zaimplementowana w ten sposób, że zwracana pamięć może zawierać dowolny typ obiektów pod warunkiem, że żądana ilość pamięci jest wystarczająco duża. Takie problemy, jak wyrównanie do granicy słowa, brane są pod uwagę wewnętrznie przez algorytm funkcji `malloc`. Zauważ, że wielkość struktury `item` jest określana za pomocą operatora `sizeof` w C, który zwraca wartość mierzoną w bajtach.

Odwrotnością `malloc` stanowi funkcja `free`, zwalniająca pamięć przydzieloną poprzednio przez algorytm `malloc`, udostępniając ją do ponownego użytku. Do funkcji `free` przekazywany jest wskaźnik, uzyskany poprzednio za pomocą wywołania funkcji `malloc`.

```
struct item *ptr;
.
.
.
ptr = (struct item *)malloc( sizeof(struct item) );
/* wykonaj pracę ... */
free( (void *)ptr );
```

Po wywołaniu w ten sposób funkcji `free` przestrzeń pamięci wskazywana przez `ptr` nie powinna być dłużej używana, ponieważ `malloc` może później ponownie przydzieleć całą tą przestrzeń lub jej część. Bardzo ważne jest przekazanie do funkcji `free` wskaźnika, który rzeczywiście został uzyskany za pomocą wywołania `malloc` lub jednej z dwóch bliźniaczych (opisanych dalej) funkcji: `calloc` lub `realloc`. Jeśli wskaźnik nie spełnia tego kryterium, niemal na pewno wyniknie poważny błąd pamięci, prowadzący do błędnego zachowania programu lub nawet katastrofalnego rzutu rdzenia. Niewłaściwe użycie funkcji `free` stanowi bardzo częsty błąd programistów.

Dwie dalsze funkcje w rodzinie `malloc` są bezpośrednio związane z przydzieleniem pamięci. Pierwszą z nich jest funkcja `calloc`.

Użycie

```
#include <stdlib.h>
void * calloc(size_t n elem, size_t nbytes);
```

Funkcja `calloc` przydziela pamięć dla tablicy `n elem` elementów, z których każdy ma wielkość `nbytes`. Zwykle jest używana w następujący sposób:

```
/* przydzieli tablicę struktur */
struct item *aptr;
.
.
.
aptr = (struct item *) calloc(nitem, sizeof(struct item));
```

W odróżnieniu od `malloc`, pamięć przydzielana przez funkcję `calloc` jest zerowana, co wymaga oczywiście dodatkowego czasu, ale może być przydatne, jeśli taka inicjacja jest wymagana.

Końcową funkcję przydziela w rodzinie `malloc` stanowi funkcja `realloc`.

Użycie

```
#include <stdlib.h>
void * realloc(void *oldptr, size_t newsize);
```

Funkcja `realloc` jest używana do zmiany wielkości bloku pamięci wskazywanego przez `oldptr`, który został otrzymany poprzednio za pomocą jednej z funkcji: `malloc`, `calloc` lub `realloc`. Funkcja `realloc` może przemieścić blok w pamięci, więc zwraca wskaźnik oznaczający jego nową początkową pozycję. Zawartość jest więc chroniona w zakresie od początku bloku do mniejszej spośród nowej i starej wielkości.

Przykład malloc: lista powiązana

W informatyce istnieje wiele typów dynamicznych struktur danych. Jeden z klasycznych przykładów stanowi lista powiązana, w której grupa identycznych obiektów jest powiązana w jednostkę logiczną. W tym podrozdziale będziemy projektować proste przykłady list powiązanych w celu pokazania zastosowań rodziny funkcji `malloc`. Zaczniemy od przyjrzenia się plikowi nagłówkowemu `list.h`:

```
/* list.h -- plik nagłówkowy dla przykładu listy powiązanej */
#include <stdio.h>
#include <stdlib.h>
```

```
/* definicja struktury podstawowej */
typedef struct list_member {
    char *m_data;
    struct list_member *m_next;
} MEMBER;

/* definicje funkcji */
MEMBER *new_member(char *);
void add_member(MEMBER **head, MEMBER *newmem);
void free_list(MEMBER **head);
void printlist(MEMBER *);
```

Instrukcja `typedef` wprowadza typ o nazwie `MEMBER`, który zawiera dwa pola. Pierwsze pole `m_data` będzie, w rzeczywistym wystąpieniu `MEMBER`, wskazywać jakiś arbitralny napis. Drugie pole `m_next` wskazuje inny obiekt typu `MEMBER`. W definicji `m_next` z przyczyn składowych musimy użyć `struct list_member *m_next` zamiast `MEMBER *m_next`.

W liście powiązanej struktur typu `MEMBER` każde pole `m_next` wskazuje na następną strukturę `MEMBER` na liście; oznacza to, że mając dany element z listy, możemy znaleźć następny, używając po prostu wskaźnika `m_next` danego elementu. Ponieważ dla każdej struktury `MEMBER` na liście istnieje jeden wskaźnik, lista może być przeglądana tylko w jednym kierunku. Taka lista jest opisywana jako **jednokierunkowa lista powiązana** (ang. *singly linked list*). Jeśli zdefiniujemy wskaźnik `m_prev`, listę będzie można także łączyć w odwrotnym kierunku i w tym przypadku będzie to **dwukierunkowa lista powiązana** (ang. *doubly linked list*).

Adres początku albo nagłówka `MEMBER` listy jest zwykle rejestrowany w specjalnym wskaźniku zadeklarowanym w sposób zbliżony do poniższego:

```
MEMBER *head = (MEMBER *)0;
```

Koniec listy został znaczony przez pustą wartość (`NULL`) w polu `m_next` ostatniej faktycznej struktury `MEMBER` na liście.

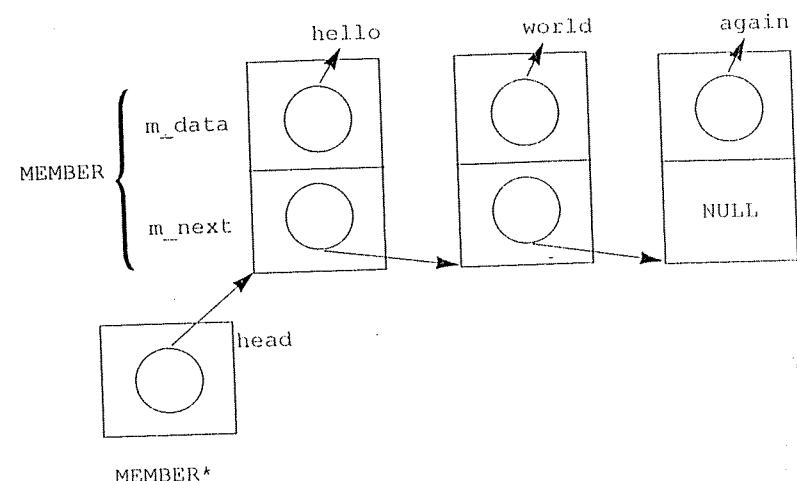
Na rysunku 12.1 pokazano prostą listę trójelementową. Jej początek jest wyznaczony przez wskaźnik o nazwie `head`.

Wprowadzimy teraz mały zestaw procedur w celu manipulowania tymi strukturami. Pierwsza funkcja, którą omówimy, ma nazwę `new_member`. Używa ona `malloc` do utworzenia wystarczającego miejsca dla struktury `MEMBER`. Zwróć uwagę, że ustawiamy wskaźnik `m_next` na wartość pustą, reprezentowaną tu jako `(MEMBER *) 0`. Trzeba to zrobić, ponieważ funkcja `malloc` nie zeruje przydzielanej pamięci i dlatego w chwili utworzenia struktury `MEMBER` pole `m_next` może zawierać fałszywy, chociaż pozornie wiarygodny adres.

```
/* new_member -- przydziela pamięć dla nowej składowej */

#include <string.h>
#include "list.h"
MEMBER * new_member(char *data)
{
    MEMBER *newmem;
    if((newmem=(MEMBER *)malloc(sizeof(MEMBER)))==(MEMBER *)0)
        fprintf(stderr, "out of memory in new_member\n");
    else
```

```
{ /* utwórz pamięć dla kopирования do niej danych */
    newmem->m_data = (char *)malloc(strlen(data)+1);
    /* skopiuj dane do struktury, zakładając, że jest to napis */
    strcpy(newmem->m_data, data);
    /* ustaw wskaźnik w strukturze jako pusty */
    newmem->m_next = (MEMBER *)0;
}
return (newmem);
```



Rysunek 12.1 Lista powiązana struktur MEMBER

Następna procedura `add_member` dodaje strukturę `MEMBER` do listy zaczynającej się w `*head`. Możesz zobaczyć, że procedura ta zawsze dodaje nową składową `MEMBER` na początku listy.

```
/* add_member -- dodaje strukturę MEMBER do listy */
#include "list.h"

void add_member(MEMBER **head, MEMBER *newmem)
{
    /* ta prosta procedura umieszcza nową składową
     * na początku listy
    */
    newmem->m_next = *head;
    *head = newmem;
}
```

Końcową procedurą narzędziową, którą omówimy, jest `free_list`. Pobiera wskaźnik umieszczony w `*head` i uwalnia pamięć używaną przez wszystkie jego

składowe struktury MEMBER. Ustawia także wskaźnik *head na wartość pustą, zapewniając, że *head nie zawiera teraz żadnej znaczącej wartości (jeśli tego nie zrobimy, wskaźnik *head może być gdzieś źle użyty).

```
/* free_list -- zwalnia całą listę */
#include "list.h"
void free_list(MEMBER **head)
{
    MEMBER *curr, *next;

    for(curr = *head; curr != (MEMBER *)0; curr = next)
    {
        next = curr->m_next;
        /* zwolnij pamięć przydzieloną dla danych */
        free((void *)curr->m_data);
        /* zwolnij pamięć przydzieloną dla struktury listy */
        free((void *)curr);
    }

    /* ustaw ponownie wskaźnik początku listy */
    *head = (MEMBER *)0;
}
```

Następujący prosty przykład zbiera wszystko razem. Tworzy on tą samą listę powiązaną, którą widzieliśmy na rysunku 12.1, a następnie usuwa ją. Zwróć uwagę na sposób, w jaki procedura printlist przegląda listę. Zawarta w niej pętla for jest typowa dla programów, które używają list powiązanych.

```
/* Program testowy dla procedur listy */
```

```
#include "list.h"
char *strings[] = { "again", "world", "Hello" };

main()
{
    MEMBER *head, *newm;
    int j;

    /* zainicjuj listę */
    head = (MEMBER *)0;

    /* dodaj składowe do listy */
    for(j = 0; j < 3; j++)
    {
        newm = new_member(strings[j]);
        add_member(&head, newm);
    }

    /* wyświetl składowe listy */
    printlist(head);

    /* usuń listę */
    free_list(head);

    /* wyświetl składowe listy */
    printlist(head);
```

```
)
```

```
/* przegląda i drukuje listę */
void printlist(MEMBER *listhead)
{
    MEMBER *m;

    printf("\nList Contents:\n");

    if(listhead == (MEMBER *)0)
        printf("\t(empty)\n");
    else
        for(m = listhead; m != (MEMBER *)0; m = m->m_next)
            printf("\t%s\n", m->m_data);
}
```

Zwróć także uwagę na sposób inicjacji listy na początku programu za pomocą ustawienia head na (MEMBER *)0. Jest to ważne, gdyż w przeciwnym przypadku lista będzie zawierała śmieci i może spowodować błędy pamięci.

Program wyświetla następujące wyjście:

```
List Contents:
Hello
world
again
```

```
List Contents:
(empty)
```

Funkcje brk i sbrk

Dla porządku powinniśmy też wspomnieć o funkcjach brk i sbrk. Są to pierwotne funkcje dynamicznego przydziału pamięci dostarczane przez Uniks. Działają one na zasadzie dostosowania wielkości segmentu danych procesu lub, dokładniej, położenia pierwszego bajtu powyżej segmentu danych procesu. Funkcja brk przemieszcza to położenie do bezwzględnego adresu, podczas gdy funkcja sbrk wykonuje przemieszczenie względne. W większości sytuacji doradzamy zdecydowanie użycie funkcji malloc i pokrewnych, a nie tych dwóch funkcji pierwotnych.

Ćwiczenie 12.1 Nasz przykład listy powiązanej może być używany do zaimplementowania stosu, gdzie ostatnia dodana składowa jest pierwszą używaną. Funkcja add_member daje nam operację push (umieszczenia na stosie); napisz operację pop (pobrania ze stosu), która usuwa pierwszy element z listy.

Ćwiczenie 12.2 Napisz program, który używa funkcji składowych rodziny malloc do przydziału pamięci dla pojedynczej liczby całkowitej, tablicy zmiennych typu float i tablicy wskaźników char.

12.3 I/O odwzorowane w pamięci i manipulacja pamięcią

Jeśli proces ma do wykonania wielką liczbę dyskowych operacji I/O, wydajność może spaść na skutek przeciążenia kopiowaniem danych z dysku do wewnętrznych buforów jądra, a następnie kopiowania tych danych do struktur zawartych w procesie użytkownika. Odwzorowane w pamięci I/O zwiększa szybkość dostępu do pliku przez odwzorowanie plików trzymanych na dysku bezpośrednio do przestrzeni pamięci procesu. Wykonują to procedury `mmap` i `munmap`. Jak widzieliśmy w przykładach dotyczących wspólnej pamięci w podrozdziale 8.3.4, manipulacja danymi w przestrzeni adresowej procesu jest najsukceszniejszą formą dostępu. (Jednak użycie na dużą skalę I/O odwzorowanego w pamięci, chociaż zwiększa szybkość dostępu do pliku, może zmniejszyć ilość pamięci dostępnej dla innych funkcji przydziału pamięci).

Procedury `mmap` i `munmap` omówimy niebawem. Wcześniej jednak przedstawimy kilka prostych procedur do manipulacji kawałkami pamięci.

Użycie

```
#include <string.h>
void * memset(void *buf, int character, size_t size);
void * memcpy(void *buf1, const void *buf2, size_t size);
void * memmove(void *buf1, const void *buf2, size_t size);
int memcmp(const void *buf1, const void *buf2, size_t size);
void * memchr(const void *buf, int character, size_t size);
```

Dla celów inicjacji może być użyta procedura `memset`, która ustawia `size` bajtów bufora `buf` na wartość `character`.

Aby wykonać bezpośrednią kopię jednego fragmentu pamięci do innego, można użyć `memcpy` lub `memmove`. Obie funkcje przemieszczają `size` bajtów pamięci z rejonu zaczynającego się w `buf2` do rejonu zaczynającego się w `buf1`. Różnica między nimi polega na tym, iż `memmove` gwarantuje, że jeśli bufore `buf1` i `buf2` zachodzą na siebie, dane podczas przemieszczania nie ulegną uszkodzeniu. Aby to zapewnić, funkcja `memmove` kopiuje najpierw zawartość `buf2` do tymczasowej tablicy, a później z tymczasowej tablicy do `buf1`.

Funkcja `memcmp` działa w tej sam sposób, co `strcmp`. Tak więc, jeśli pierwsze `size` bajtów bufora `buf1` jest takich samych, jak pierwsze `size` bajtów `buf2`, `memcmp` zwróci wartość 0.

Funkcja `memchr` przeszukuje pierwsze `size` bajtów bufora `buf` i zwraca adres pierwszego wystąpienia znaku `character` albo `NULL`. Zasadniczo inne procedury zwracają wartość pierwszego wskaźnika ze swojej listy parametrów.

Funkcje systemowe `mmap` i `munmap`

Zobaczmy teraz, jak ustawić i zakończyć początkowe odwzorowanie. I/O odwzorowane w pamięci jest implementowane za pomocą funkcji systemowej `mmap`. Pracuje ona ze stronami pamięci i dlatego plik musi być wyrównany w pamięci do granicy strony. Funkcja `mmap` jest definiowana następująco:

Użycie

```
#include <sys/mman.h>
void * mmap(void *address, size_t length, int protection,
           int flags, int filedes, off_t offset);
```

Plik, o którym mowa, musi być najpierw otwarty za pomocą funkcji systemowej `open`. Zwrócony przez `open` deskryptor pliku jest używany jako argument `filedes` w wywołaniu `mmap`.

Pierwszy argument `mmap`, `address`, pozwala programistie określić, gdzie w przestrzeni adresowej procesu powinno się zacząć odwzorowanie. Jak widzieliśmy przy `shmat` w podrozdziale 8.3.4, zdolność określenia adresu oznacza, że programista musi wiedzieć, jak rozłożona jest pamięć procesu. Oczywiście znacznie bezpieczniej (i bardziej przenośnie) jest pozwolić systemowi samemu wybrać adres początkowy, ustawiając parametr `address` na wartość 0. Wartość zwracana przez `mmap` stanowi wtedy początkowy adres odwzorowania. W przypadku błędu `mmap` zwraca wartość (`void *`) -1.

Parametr `offset` określa, gdzie w pliku rozpoczyna się odwzorowanie. W większości zastosowań programista zechce odwzorować w pamięci cały plik i dlatego `offset` będzie ustawiony na wartość 0 – co oznacza początek pliku. Jeśli `offset` jest różny od 0, musi być wielokrotnością rozmiaru strony pamięci.

Liczba bajtów pliku odwzorowywaną począwszy od `offset` określa parametr `length`. Jeśli `length` nie jest wielokrotnością rozmiaru strony pamięci, faktyczne `length` bajtów wygląda jak oczekiwaliśmy, natomiast pozostała część strony zostaje wypełniona zerami.

Parametr `protection` określa, czy dane w przestrzeni adresowej mogą być odczytywane, zapisywane, wykonywane lub udostępniane. Może on przybierać jedną lub więcej z następujących wartości, zdefiniowanych w `<sys/mman.h>`:

<code>PROT_READ</code>	
<code>PROT_WRITE</code>	
<code>PROT_EXEC</code>	
<code>PROT_NONE</code>	

Pamięć może być odczytywana.
Pamięć może być zapisywana.
Pamięć może być wykonywana.
Pamięć nie jest dostępna.

Jednak wartość `protect` musi być zgodna ze sposobem, w jaki plik został otwarty. Parametr `flags` wpływa na sposób, w jaki zapis do odwzorowanego pliku jest widziany przez inne procesy. Najbardziej przydatne są następujące wartości:

MAP_SHARED Sprawia, że dowolne zmiany regionu są widziane przez inne procesy odwzorowujące plik, a modyfikacje są zapisywane do faktycznego pliku.

MAP_PRIVATE Modyfikacje regionu nie są widziane przez inne procesy i nie są zapisywane do faktycznego pliku.

Gdy plik został już odwzorowany, może być dostępny bezpośrednio przez lokale pamięci. Kiedy lokacja pamięci jest odczytywana, odpowiadające bajty pliku stają się widoczne.

Kiedy proces kończy się, plik przestaje być odwzorowany. Jednak, aby zakończyć odwzorowanie przed końcem programu, trzeba użyć funkcji systemowej munmap.

Użycie

```
#include <sys/mman.h>
int munmap(void *address, size_t length);
```

Jeśli został ustawiony znacznik MAP_SHARED, plik jest aktualizowany z wszystkimi pozostawionymi zmianami. Jeśli natomiast został ustawiony znacznik MAP_PRIVATE, wszystkie modyfikacje są usuwane.

Jednak uważaj, ponieważ ta funkcja kończy tylko odwzorowanie pliku w pamięci, a nie zamknie pliku. Zamknąć plik trzeba w dalszym ciągu za pomocą funkcji systemowej close.

Następujący przykład stanowi powtórzenie programu copyfile, który widzieliśmy w podrozdziale 11.4. Program otwiera plik wejściowy i wyjściowy i kopiuje zawartość jednego do drugiego. W celu zachowania przejrzystości opuściliśmy obsługę błędów.

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>

main(int argc, char **argv)
{
    int input, output;
    size_t filesize;
    void *source, *target;
    char endchar = '\0';

    /* sprawdź, czy przekazano prawidłową liczbę parametrów */
    if(argc != 3)
    {
        fprintf(stderr, "usage: copyfile source target\n");
        exit(1);
    }

    /* otwórz plik wejściowy i wyjściowy */
```

```
if((input = open(argv[1], O_RDONLY)) == -1)
{
    fprintf(stderr, "Error opening file %s\n", argv[1]);
    exit(1);
}

if((output = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0666)) == -1)
{
    close(input);
    fprintf(stderr, "Error opening file %s\n", argv[2]);
    exit(2);
}

/* utwórz drugi plik o tej samej wielkości, co pierwszy */
filesize = lseek(input, 0, SEEK_END);
lseek(output, filesize - 1, SEEK_SET);
write(output, &endchar, 1);

/* odwzoruj w pamięci plik wejściowy i wyjściowy */
if((source = mmap(0, filesize, PROT_READ, MAP_SHARED, input,
                  0)) == (void *)-1)
{
    fprintf(stderr, "Error mapping first file\n");
    exit(1);
}
if((target = mmap(0, filesize, PROT_WRITE, MAP_SHARED, output,
                  0)) == (void *)-1)
{
    fprintf(stderr, "Error mapping second file\n");
    exit(2);
}

/* kopiuj */
memcpy(target, source, filesize);

/* zakończ odwzorowanie obu plików */
munmap(source, filesize);
munmap(target, filesize);

/* zamknij oba pliki */
close(input);
close(output);

exit(0);
}
```

Oczywiście zakończenie odwzorowania i zamknięcie plików nastąpi także przy naturalnym zakończeniu programu, jednak do kompletu dodaliśmy te instrukcje.

12.4 Czas

Unix dostarcza garść procedur służących do poznawania i ustawiania systemowego pojęcia czasu. Czas jest mierzony jako liczba sekund, które uplynęły od dnia 1 stycznia 1970 roku, godziny 00:00:00 czasu GMT i musi być przechowywany co najmniej w zmiennej typu `long integer`. Dlatego system dostarcza typu `time_t`, który jest zdefiniowany w `<sys/types.h>`.

Najbardziej podstawową funkcją jest `time`, funkcja systemowa zwracająca bieżący czas w standardowym formacie Uniksa.

Użycie

```
#include <time.h>
time_t time(time_t *now);
```

Po wywołaniu tej funkcji `now` będzie zawierać systemowe pojęcie czasu. Funkcja `time` umieszcza także bieżący czas w zwracanej przez siebie wartości, ale zwykle jest to ignorowane.

Trudno jest myśleć w kategoriach wielkiej liczby sekund, więc Unix dostarcza szeregu procedur bibliotecznych dla przetworzenia informacji o czasie w bardziej zrozumiałą postać. Podstawową z tych procedur jest `ctime`, która na podstawie wartości zwracanej przez wywołanie funkcji `time` generuje 26-znakowy napis. Na przykład:

```
#include <time.h>

main()
{
    time_t timeval;
    time(&timeval);
    printf("The time is %s\n", ctime(&timeval));
    exit(0);
}
```

wyświetli następujący wynik:

The time is Tue Mar 18 00:17:06 1998

który pokazuje, jak długo pracujemy nad tą książką (jednak nie tak długo, jak nad pierwszym wydaniem!).¹

Z `ctime` związany jest szereg procedur, które używają struktur `struct tm`. Wzorzec `tm` został zdefiniowany w pliku nagłówkowym `<time.h>` i zawiera następujące składowe:

```
int tm_sec;      /* sekundy */
int tm_min;      /* minuty */
int tm_hour;     /* godziny, od 0 do 24 */
int tm_mday;     /* dni miesiąca, od 1 do 31 */
int tm_mon;      /* miesiące, od 0 do 11 */
int tm_year;     /* rok minus 1900 */
int tm_wday;     /* dzień tygodnia, niedziela = 0 */
int tm_yday;     /* dzień roku, 0-365 */
int tm_isdst;    /* znacznik czasu letniego, tylko dla USA */
```

Cel każdej składowej powinien być oczywisty. Poniżej pokazano kilka procedur używających tej struktury.

Użycie

```
#include <time.h>
struct tm * localtime(const time_t *timeval);
struct tm * gmtime(const time_t *timeval);
char * asctime(const struct tm *ptr);
time_t mktime(struct tm *ptr);
double difftime(time_t time1, time_t time2);
```

Procedury `localtime` i `gmtime` pobierają wartość otrzymaną uprzednio od funkcji `time` i zamieniają na strukturę `tm`, zwracając odpowiednio lokalną i GMT wersję czasu. Na przykład:

```
/* tm -- pokaż strukturę tm */

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

main()
{
    time_t t;
    struct tm *tp;
    /* pobierz czas z systemu */    time(&t);
    /* pobierz strukturę tm */
    tp = localtime(&t);
    printf("Time %d:%d:%d\n", tp->tm_hour, tp->tm_min, tp->tm_sec);
    exit(0);
}
```

daje następujący wynik:

Time 1:13:23

Procedura `asctime` zamienia strukturę `tm` w napis podobny do tworzonego przez `ctime`, a procedura `mktme` zamienia strukturę `tm` na równoważną jej liczbę sekund. Procedura `difftime` zwraca różnicę w sekundach między dwoma wartościami czasu kalendarzowego.

¹ Por. przyp. za s. VII.

Ćwiczenie 12.3 Napisz własną wersję procedury `asctime`.

Ćwiczenie 12.4 Napisz funkcję `weekday`, która zwraca 1, jeśli dzień jest roboczy, a zero w przeciwnym przypadku. Napisz funkcję odwrotną do niej o nazwie `weekend`. Funkcje te powinny być w pewien sposób konfigurowalne.

Ćwiczenie 12.5 Napisz procedury, zwracające różnicę między dwoma datami otrzymanymi z `time` w dniach, miesiącach, latach i sekundach. Pamiętaj o latach przestępnych!

12.5 Napisy i manipulacja znakami

Biblioteki Uniksów są bogate w funkcje, które manipulują napisami albo danymi znakowymi. Są one wystarczająco przydatne, aby usprawiedliwić krótkie omówienie.

12.5.1 Rodzina funkcji string

Używaliśmy już kilku z tych dobrze znanych procedur, na przykład `strcat` i `strcpy`. Oto obszerniejsza lista:

Użycie

```
#include <string.h>

char * strcat(char *s1, const char *s2);
char * strncat(char *s1, const char *s2, size_t length);

int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t length);
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t length);

char * strcpy(char *s1, const char *s2);
char * strncpy(char *s1, const char *s2, size_t length);
char * strdup(const char *s1);

size_t strlen(const char *s1);

char * strchr(const char *s1, int c);
char * strrchr(const char *s1, int c);
char * strstr(const char *s1, const char *s2);

char * strpbrk(const char *s1, const char *s2);
size_t strspn(const char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);

char * strtok(char *s1, const char *s2); /* pierwsze wywołanie */
char * strtok(NULL, const char *s2); /* kolejne wywołania */
```

Procedura `strcat` dołącza napis `s2` do końca napisu `s1`. Procedura `strncat` wykonuje to samo, ale dołącza co najwyżej `length` znaków. Obie procedury zwracają wskaźnik do napisu `s1`. Oto przykład użycia procedury `strcat`:

Jeśli `fileprefix` początkowo zawiera napis "file", po zakończeniu procedury będzie zawierać napis "file.dat". Należy zwrócić uwagę, że procedura `strcat` zmienia napis wskazywany przez jej pierwszy argument. Właściwość ta jest wspólna dla procedur `strncat`, `strcpy`, `strncpy` i `strtok`. Ponieważ procedury w C nie znają wielkości przekazywanych do nich tablic, programista musi upewnić się, że pierwszy argument tych procedur będzie wystarczająco duży do zapamiętania wyniku żądanej operacji.

Procedura `strcmp` porównuje dwa napisy `s1` i `s2`, zwracając zmienną stanowiącą wskazówkę. Jeśli zwracana wartość jest dodatnia, oznacza to, że napis `s1` jest leksykograficznie większy niż `s2`, stosownie do kolejności zestawu znaków ASCII. Wartość ujemna oznacza, że napis `s1` jest leksykograficznie mniejszy niż `s2`. Jeśli zwracana jest wartość zerowa, oba napisy są identyczne. Procedura `strncmp` jest podobna, ale porównuje co najwyżej `length` znaków. Funkcje `strcasecmp` i `strncasecmp` wykonują dokładnie te same porównania, ale ignorują różnice w wielkości liter. Oto przykład zastosowania procedury `strcmp`:

```
if(strcmp(token, "print") == 0)
{
    /* przetwórz słowo kluczowe print */
}
```

Procedura `strcpy` jest blisko związana ze `strcat`. Kopiuje ona zawartość `s2` do `s1`, niszcząc oryginalną zawartość `s1`. Procedura `strncpy` kopiuje dokładnie `length` znaków, obcinając lub dodając znaki zerowe w miarę potrzeby (co może znaczyć, że `s1` nie jest zakończony znakiem zerowym). Procedura `strdup` zwraca wskaźnik do nowej kopii napisu `s1`. Zwrócony wskaźnik może być przekazany do funkcji `free`, ponieważ pamięć pochodzi z wywołania funkcji `malloc`.

Procedura `strlen` po prostu zwraca długość napisu `s1`. Innymi słowy, zwraca liczbę znaków w `s1`, nie licząc końcowego znaku zerowego (`null`).

Procedura `strchr` zwraca wskaźnik do pierwszego wystąpienia znaku `c` (w rzeczywistości zadeklarowanego jako `int`) w napisie `s1` lub `NULL`, jeśli nie znalazła dopasowania. Procedura `strrchr` wykonuje to samo dla ostatniego wystąpienia znaku `c`. Procedura `strrstr` zwraca wskaźnik do pierwszego wystąpienia napisu `s2` w napisie `s1`. Używaliśmy `strrchr` w rozdziale 4 do usunięcia nazwy ścieżki z początkowej części nazwy pliku:

```
/* wycina część z nazwy ścieżki */
filename = strrchr(pathname, '/');
```

Procedura `strpbrk` zwraca wskaźnik do pierwszego wystąpienia w `s1` dowolnego ze znaków z napisu `s2`. Jeśli nie ma żadnego dopasowania, zwracany jest wskaźnik `NULL`.

Procedura `strspn` zwraca długość tej części napisu `s1`, poczynając od pierwszego znaku `s1`, która składa się całkowicie ze znaków z napisu `s2`. Procedura `strcspn` zwraca długość początkowego segmentu `s1`, nie zawierającego żadnego znaku z `s2`.

Końcowa procedura `strtok` pozwala programowi rozdzielić napis `s1` na indywidualne jednostki leksykalne – leksemy (ang. *tokens*). Napis `s2` zawiera tu znaki,

które mogą rozdеляć leksemu (na przykład: spacje, tabulacje i znaki nowego wiersza). Pierwsze wywołanie, z pierwszym argumentem ustawionym na `s1`, powoduje, że `strtok` zapamiętuje napis. Zostanie wtedy zwrócony wskaźnik do pierwszego leksemu. Kolejne wywołania, z pierwszym argumentem ustawionym na wartość `NULL`, dają dalsze leksemu z napisu `s1`. Gdy nie pozostały już żadne leksemu, zostanie zwrócony wskaźnik pusty.

12.5.2 Konwersja napisu na liczbę

Standard ANSI C dostarcza dwóch zestawów funkcji do zamiany napisów na liczby:

Użycie

```
#include <stdlib.h>
/* zamiana napisu na liczbę typu integer */
long int strtol(const char *str, char **endptr, int base);
long int atol(const char *str);
int atoi(const char *str);
/* zamiana napisu na liczbę typu double */
double strtod(const char *str, char **endptr);
double atof(const char *str);
```

Funkcje `atoi`, `atol` i `atof` pobierają stałą napisową i konwertują odpowiednio na liczbę typu `integer`, `long` lub `double`. Jednak te funkcje zostały teraz zastąpione przez `strtol` i `strtod`.

Funkcje `strtod` i `strtol` są znacznie skuteczniejsze w działaniu. Obie pobierają napis `str` i usuwają każdy odstęp z początku napisu i każdy nierozpoznany znak z końca napisu (włącznie ze znakiem zerowym). Następnie zapamiętują napis do konwersji w `endptr`, dopóki `endptr` nie jest pusty. Ostatni parametr `strtol`, `base`, może być ustawiony na dowolną wartość pomiędzy 0 i 36 i napis będzie zmieniony na liczbę o tej określonej podstawie systemu liczbowego.

12.5.3 Znaki: zatwierdzanie i konwersja

Unix dostarcza dwa użyteczne zestawy makroinstrukcji i funkcji do manipulowania znakami; oba są zdefiniowane w pliku nagłówkowym `<ctype.h>`. Pierwszy zestaw, zgrupowany pod nagłówkiem `ctype`, przeznaczony jest do zatwierdzania pojedynczych znaków. Są to wszystko makroinstrukcje boole'owskie, które zwracają 1 (`true`), jeśli warunek jest prawdziwy, i 0 (`false`), jeśli jest fałszywy. Na przykład `isalpha` sprawdza, czy znak jest literą, to znaczy znajduje się w zakresie `a-z` lub `A-Z`, czy też nie:

```
#include <ctype.h>
int c;
.
.
```

```
/* "isalpha" jest makroinstrukcją ctype */
if( isalpha(c) )
{
    /* przetwórz znak alfabetyczny, tzn. literę */
}
else
    warn("Character is not a letter");
```

Zwróci uwagę, że zmienna `c` jest zadeklarowana jako `int`. Oto pełna lista makroinstrukcji `ctype`:

<code>isalpha(c)</code>	czy <code>c</code> jest literą?
<code>isupper(c)</code>	czy <code>c</code> jest wielką literą?
<code>islower(c)</code>	czy <code>c</code> jest małą literą?
<code>isdigit(c)</code>	czy <code>c</code> jest cyfrą (0-9)?
<code>isxdigit(c)</code>	czy <code>c</code> jest cyfrą szesnastkową?
<code>isalnum(c)</code>	czy <code>c</code> jest literą lub cyfrą?
<code>isspace(c)</code>	czy <code>c</code> jest odstępem (białą spacią); to znaczy spacią, tabulacją, powrotem karetki, znakiem nowego wiersza, wysunięcia strony lub tabulacji pionowej?
<code>ispunct(c)</code>	czy <code>c</code> jest znakiem przestankowym?
<code>isprint(c)</code>	czy <code>c</code> jest znakiem drukowalnym? W zestawie znaków ASCII oznacza to dowolny znak w zakresie od spacji (040) do tyldy (~ albo 0176).
<code>isgraph(c)</code>	czy <code>c</code> jest znakiem drukowalnym, ale nie spacią?
<code>iscntrl(c)</code>	czy <code>c</code> jest znakiem sterującym? Jako znak sterujący liczony jest znak kasowania ASCII, jak również wszystkie znaki z wartością liczbową mniejszą niż 040.
<code>isascii(c)</code>	czy <code>c</code> jest w ogóle w zestawie znaków ASCII? Zwróci uwagę, że wartość całkowita przekazywana do dowolnej z innych procedur <code>ctype</code> musi spełniać ten test, z wyjątkiem EOF ze <code><stdio.h></code> (ten wyjątek pozwala na używanie makroinstrukcji <code>ctype</code> z funkcją <code>getc</code> itd.).

Inny zestaw narzędzi opartych na znakach przeznaczony jest do prostej translacji znaków. Na przykład funkcja `tolower` zamienia za pomocą zwracanej wartości wielkie litery na ich małe odpowiedniki.

```
#include <ctype.h>
int newc, c;
.
.
/* zamienia wielkie litery na małe */
/* np.: odwzorowuje A na a */
newc = tolower(c);
```

Jeśli `c` jest wielką literą, zostaje zamieniona na małą literę. W przeciwnym przypadku pozostaje niezmieniona. Oto inne procedury i makroinstrukcje (które możesz znaleźć w swoim podręczniku pod nagłówkiem `conv`):

<code>toupper(c)</code>	Funkcja, która zamienia <code>c</code> na wielką literę, jeśli jest ona znakiem małej litery. Inaczej pozostawia <code>c</code> bez zmian.
<code>toascii(c)</code>	Makroinstrukcja, która zamienia wartość całkowitą nie-ASCII na wartość ASCII przez usunięcie bitów nie-ASCII.
<code>_toupper(c)</code>	Szybka wersja <code>toupper</code> w postaci makroinstrukcji, która nie wykonuje żadnego sprawdzania, więc musi mieć przekazany znak małej litery.
<code>tolower(c)</code>	Szybka wersja <code>tolower</code> w postaci makroinstrukcji, z podobnymi ograniczeniami jak <code>_toupper</code> .

12.6 Inne przydatne funkcje

Skoncentrowaliśmy się w tej książce na tych funkcjach i procedurach, które, jak sądzimy, dają przydatne podstawy programowania systemowego w Uniksie. Jeśli dopiero zaczynasz, pomogą ci one rozwiązać wielką liczbę problemów. Oczywiście Unix jest znacznie bogatszy i zawiera wiele innych procedur, które oferują wyspecjalizowaną funkcjonalność. Ten ostatni podrozdział ma na celu zwrócenie na nie twojej uwagi. Dalsze szczegóły znajdziesz w lokalnym podręczniku swojego systemu.

12.6.1 Więcej o gniazdach

Gniazda są bardzo skutecznym i popularnym sposobem komunikacji między procedurami i komputerami; rozdział 10 dostarczył krótkiego wprowadzenia. Jeśli chcesz wiedzieć więcej, powinieneś przeczytać specjalistyczne książki. W pierwszej kolejności możesz zbadać następujące procedury, które podają informacje o środowisku sieciowym:

<code>gethostent</code>	<code>getservbyname</code>
<code>gethostbyaddr</code>	<code>getservbyport</code>
<code>gethostbyname</code>	<code>getservent</code>

12.6.2 Wątki

Wątki są lżejszą wersją procesu – a systemy je obsługujące pozwalają prowadzić w istniejącym procesie wiele wątków, udostępniając wszystkie dane. Wątki mogą być przydatne dla uzyskania optymalnej wydajności w bardzo wyspecjalizowanej klasie problemów – ale ogólnie są złożone w użyciu. Zwykle wystarcza model procesu Uniksa. Wiele wersji Uniksa obsługuje różne podejście do wątków, ale standardowe modele są teraz włączone w dokumentację *POSIX* i *XSI Version 5*. Spróbuj poszukać w swoim podręczniku procedur z przedrostkiem `pthread_`.

12.6.3 Rozszerzenia czasu rzeczywistego

Najnowsza praca *POSIX* dodała pewne opcjonalne rozszerzenia czasu rzeczywistego. I znowu, jest to ograniczony i złożony obszar, a rdzeń Uniksa ma wystarczającą siłę dla większości typowych problemów. Specjalne własności czasu rzeczywistego zawierają:

- kolejkowanie sygnałów i dodatkowe własności sygnałów (patrz `sigwaitinfo`, `sigtimedwait`, `sigqueue`)
- sterowanie priorytetem/planowaniem (szukaj procedur zaczynających się od `sched_`)
- więcej własności czasomierzy, asynchronicznych i synchronicznych operacji I/O
- alternatywy omawianych przez nas interfejsów przekazywania komunikatorów, semaforów i wspólnej pamięci (spróbuj poszukać procedur zaczynających się od `mq_`, `sem_` i `shm_`)

12.6.4 Śledzenie działania lokalnego systemu

Omówiliśmy już wiele przydatnych do tego celu procedur (takich jak `pathconf`). Inne dostępne ułatwienia zawierają:

<code>sysconf</code>	Daje dostęp do ograniczeń systemowych i parametrów konfiguracyjnych, zawartych w <code><limits.h></code> i <code><unistd.h></code> .
<code>uname</code>	Zwraca wskaźnik do struktury <code>utsname</code> , zawierającej nazwę systemu, nazwę węzła, która może być używana przez system przy komunikacji sieciowej oraz dane wydania i wersji dla samego Uniksa.
<code>getpwent</code>	Ta rodzina procedur pozwala na dostęp do danych z pliku hasel <code>/etc/passwd</code> . Wszystkie poniższe funkcje zwracają wskaźnik do struktury <code>passwd</code> , która jest zdefiniowana w <code><pwd.h></code> . Mogą być wykonywane następujące wywołania: <code>getpwnam(const char *username);</code> <code>getpwuid(uid_t uid);</code> <code>getpwent(void);</code>
<code>getgrent</code>	Ta rodzina procedur jest związana z dostępem do pliku grupy <code>/etc/group</code> .
<code>getrlimit</code>	Daje dostęp do ograniczeń zasobów systemowych, takich jak pamięć lub przestrzeń plików.
<code>getlogin,</code> <code>cuserid</code>	Podaje nazwę rejestracyjną dla bieżącego procesu.

12.6.5 Umiedzynarodowienie

Wiele wersji Uniksa obsługuje różne międzynarodowe „przyprawy” – uwzględnia różnice w języku, sekwencje zestawień, symbole monetarne, formaty numeryczne itd. Spróbuj poszukać takich procedur, jak `setlocale` lub `catopen`. Twój lokalny podręcznik może zawierać szczegóły odnośnych parametrów środowiska pod nagłówkiem `environ`.

12.6.6 Procedury matematyczne

Unix dostarcza olbrzymią bibliotekę procedur matematycznych dla programistów naukowych lub technicznych. Niektóre z tych procedur powinny być używane w połączeniu z plikiem nagłówkowym `<math.h>`, który zawiera definicje funkcji, kilka ważnych stałych (takich jak `e` i `π`) i definicje struktur wykorzystywanych przy obsłudze błędów. Aby użyć większości poniżej wymienionych procedur, będziesz musiał dodać do swojego programu bibliotekę matematyczną Uniksa za pomocą wiersza poleceń

```
cc -o mathprog mathprog.c -lm
```

Procedury występują w podręcznikach Uniksa i *XSI* pod następującymi nagłówkami:

<code>abs</code>	Zwraca wartość bezwzględną liczby całkowitej. Stanowi część standardowej biblioteki C, więc nie musisz dodać biblioteki matematycznej.
<code>cbrt</code>	Znajduje pierwiastek sześcienny liczby.
<code>div</code>	Oblicza iloraz i resztę z dzielenia całkowitego.
<code>drand48</code>	Zestaw funkcji do generacji liczb pseudolosowych (zobacz też prostszą wersję – <code>rand</code>).
<code>erf</code>	Funkcja matematyczna błędu (która nie powinna być mylona z obsługą błędu w sensie programowym).
<code>exp/log</code>	Zestaw funkcji wykładniczych i logarytmicznych.
<code>floor</code>	Procedura obcinania lub uzyskiwania wartości bezwzględnej wyrażeń zmennoprzecinkowych (zobacz też <code>ceil</code>).
<code>frexp</code>	Procedura manipulacji częścią liczb zmennoprzecinkowych.
<code>gamma</code>	Wersja funkcji <code>log gamma</code> .
<code>hypot</code>	Euklidesowa funkcja odległości. Przydatna dla pokazania dzieciom, że komputery są czasem przydatne.
<code>sinh</code>	Nagłówek ten zawiera funkcje <code>sinh</code> , <code>cosh</code> i <code>tanh</code> .
<code>sqrt</code>	Znajduje pierwiastek kwadratowy liczby.
<code>trig</code>	Wskazuje grupę funkcji trygonometrycznych: <code>sin</code> , <code>cos</code> , <code>tan</code> , <code>asin</code> , <code>acos</code> , <code>atan</code> i <code>atan2</code> . Mogą mieć one własne indywidualne pozycje w podręczniku.

DODATEK A

Kody błędów errno i związane z nimi komunikaty

A.1 Wprowadzenie

Jak widzieliśmy w rozdziale 2, Unix dostarcza zestaw standardowych kodów błędów i komunikatów opisujących błędy, które mogą być zwracane przez funkcje systemowe. Dokładniej każdy błąd funkcji systemowej ma swój numer błędu, kod mnemoniczny i napis komunikatu. Może on być używany po włączeniu pliku nagłówkowego `errno.h`.

Jeśli zdarzy się błąd, funkcja systemowa ustawia odpowiednią wartość w `errno`. W niemal wszystkich przypadkach funkcja systemowa będzie też zwracać `-1` do procesu wywołującego, sygnalizując powstanie błędu. Można wtedy porównywać wartość `errno` z kodami mnemonicznymi zdefiniowanymi w `errno.h`. Na przykład:

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

pid_t pid;
.

.

if((pid = fork()) == -1)
{
    if(errno == EAGAIN)
        fprintf(stderr, "process limit reached, try again\n");
    else
        fprintf(stderr, "other error\n");
}
```

Zewnętrzna tablica `sys_errlist` stanowi tablicę komunikatów błędów drukowanych przez procedurę `perror`. Kiedy wymagany jest komunikat stosowny do błędnej sytuacji, wartość `errno` może być używana jako indeks tablicy. Zewnętrzna wartość całkowita `sys_nerr` podaje bieżącą wielkość tablicy `sys_errlist`. Zmienna `errno` zawsze powinna być porównana z `sys_nerr` przed użyciem jej jako indeksu tablicy `sys_errlist`, ponieważ mogą być dodane nowe numery błędów, zanim tablica napisów zostanie powiększona.

A.2 Lista kodów błędów i komunikatów

Poniżej zamieszczamy listę komunikatów o błędach funkcji systemowych. Jest ona oparta na informacji z Issue 4.2, standardowego interfejsu systemu X/Open. Każda pozycja podaje kod mnemoniczny z `errno.h`, systemowy komunikat o błędzie zawarty w `sys_errlist` i krótki opis. Zwróć uwagę, że tekst komunikatu błędu może zmieniać się zależnie od ustawienia kategorii `LC_MESSAGES` w bieżącym `locale` (co jest związane z międzynarodowym użyciem Uniksa).

E2BIG	<i>Argument list too long.</i> (Zbyt dłuża lista argumentów). Często oznacza to, że do wywołania funkcji <code>exec</code> została przekazana zbyt dłuża lista argumentów (w znaczeniu całkowitej liczby bajtów).
EACCES	<i>Permission denied.</i> (Brak uprawnień). Wystąpił błąd praw dostępu do pliku. Może wystąpić przy funkcjach systemowych <code>open</code> , <code>link</code> , <code>creat</code> i podobnych. Może być także wygenerowany przez funkcję <code>exec</code> , jeśli prawo wykonania nie jest ustawione.
EADDRINUSE	<i>Address in use.</i> (Adres jest używany). Oznacza to, że adres, żądany przez programistę, jest już używany.
EADDRNOTAVAIL	<i>Address not available.</i> (Adres jest niedostępny). Może to się zdarzyć, jeśli programista żąda adresu używanego już przez proces.
EAFNOSUPPORT	<i>Address family not supported.</i> (Rodzina adresów nie jest obsługiwana). Przy użyciu interfejsu funkcji gniazda określona została rodzina adresów, która nie jest obsługiwana przez ten system.
EAGAIN	<i>Resource temporarily unavailable, try again later.</i> (Zasób chwilowo niedostępny, spróbuj ponownie później). Zwykle oznacza to, że szczególna tablica systemowa jest pełna. Może być generowany przez funkcję <code>fork</code> (zbyt dużo procesów) i funkcje IPC (zbyt wiele szczególnych typów obiektów IPC).
EALREADY	<i>Connection already in progress.</i> (Połączanie już występuje). Oznacza to, że połączanie, które jest próbowane na gnieździe, zostało odrzucone, ponieważ gniazdo już pracuje z takim żądaniem.
EBADF	<i>Bad file descriptor.</i> (Niewłaściwy deskryptor pliku). Oznacza to, że albo deskryptor pliku nie reprezentuje otwartego pliku, lub że tryb dostępu, to znaczy tylko-do-odczytu, tylko-do-zapisu, nie pozwala na żądaną operację. Generowany przez wiele funkcji, włącznie z <code>read</code> i <code>write</code> .
EBADMSG	<i>Bad message.</i> (Niewłaściwy komunikat). Jest generowany, jeśli funkcja systemowa otrzyma komunikat, którego nie może odczytać. Na przykład jeśli operacja <code>read</code> była wykonywana

Dodatek A: Kody błędów errno i związane z nimi komunikaty

EBUSY	na głowie STREAM i otrzymała komunikat kontrolny STREAM zamiast komunikatu danych. <i>Device or resource busy.</i> (Urządzenie lub zasób zajęte). Na przykład może być generowany, kiedy proces próbuje <code>rmdir</code> katalogu, który jest używany przez inny proces.
ECHILD	<i>No child process.</i> (Nie ma procesu potomnego). Zostały wywołane funkcje <code>wait</code> lub <code>waitpid</code> , ale żaden odpowiedni proces potomny nie istnieje.
ECONNABORTED	<i>Connection aborted.</i> (Połączanie przerwane). Połączanie sieciowe zostało z nieznanych powodów przerwane.
ECONNREFUSED	<i>Connection refused.</i> (Połączanie odmówione). Kolejka żądań jest pełna lub żaden proces nie nasłuchuje.
ECONNRESET	<i>Connection reset.</i> (Połączanie przywrócone do stanu początkowego). Połączanie zostało zamknięte przez inny proces.
EDEADLK	<i>Resource deadlock would occur.</i> (Może wystąpić zakleszczenie zasobu). Oznacza to, że wywołanie funkcji, jeśli będzie możliwe, spowoduje zakleszczenie (innymi słowy położenie, gdy dwa procesy są zawieszone i każdy oczekuje na działanie drugiego). Ten błąd może być ustawiony przez funkcje <code>fentl</code> i <code>lockf</code> .
EDESTADDRREQ	<i>Destination request required.</i> (Wymagane żądanie przeznaczenia). Wymagany adres został pominięty przez operację gniazda.
EDOM	<i>Domain error.</i> (Błąd dziedziny). Błąd pakietu matematycznego, który oznacza, że argument funkcji jest poza dziedziną tej funkcji. Między innymi może być ustawiony przez funkcje <code>trig</code> , <code>exp</code> i <code>gamma</code> .
EDQUOT	<i>Reserved.</i> (Zarezerwowany).
EEXIST	<i>File exists.</i> (Plik istnieje). Wskazuje, że istnieje plik, który przeszkadza w wykonaniu operacji. Może być ustawiony przez funkcje <code>link</code> , <code>mkdir</code> , <code>mknod</code> , <code>shmget</code> i <code>open</code> .
EFAULT	<i>Bad address.</i> (Niewłaściwy adres). Generowany przez system po defekcie sprzętu ochrony pamięci. Zwykle oznacza, że wyszczególniony został absurdalny adres. Zdolność systemów do generowania tego błędu zmienia się znacznie.
EFBIG	<i>File too large.</i> (Zbyt duży plik). Została wykonana próba rozszerzenia pliku poza granicę wielkości pliku dla procesu (ustawioną przez <code>ulimit</code>) lub maksymalnej wielkości pliku w systemie.
EHOSTUNREACH	<i>Host is unreachable.</i> (Host jest nieosiągalny). Generowany przez sieć, jeżeli host jest wyłączony lub nieosiągalny przez ruter.
EIDRM	<i>Identifier removed.</i> (Usunięty identyfikator). Wskazuje, że identyfikator IPC, na przykład id wspólnej pamięci, został usunięty za pomocą polecenia <code>ipcrm</code> .

EILSEQ	<i>Illegal byte sequence.</i> (Nielegalna sekwencja bajtów). Znak nie odpowiada niczemu ważnemu. Może być generowany przez wywołanie funkcji <code>fprintf</code> i <code>ffscanf</code> .
EINPROGRESS	<i>Connection in progress.</i> (Połączanie w trakcie). Oznacza, że wywołane połączenie będzie zablokowane, dopóki gniazdo nie będzie gotowe do zaakceptowania. Aby to się zdarzyło, dla gniazda powinien być ustawiony znacznik <code>O_NONBLOCK</code> .
EINTR	<i>Interrupted function call.</i> (Przerwane wywołanie funkcji). Zwracany, jeśli sygnał został przechwycony w trakcie, gdy program wykonuje wywołanie funkcji systemowej. (Dotyczy tylko niektórych funkcji - sprawdź w lokalnej dokumentacji).
EINVAL	<i>Invalid argument.</i> (Nieważny argument). Oznacza po prostu, że do funkcji systemowej został przekazany nieważny parametr albo zestaw parametrów. Może być generowany przez <code>fcntl</code> , <code>sigaction</code> i kilka innych procedur IPC. Może być też ustawiany przez procedury matematyczne.
EIO	<i>I/O error.</i> (Błąd I/O). Podczas operacji I/O zdarzył się błąd fizyczny.
EISCONN	<i>Socked is connected.</i> (Gniazdo jest połączone). Żądane gniazdo ma już połączenie.
EISDIR	<i>Is a directory.</i> (To jest katalog). Była wykonana próba otwarcia katalogu do zapisu. Ten błąd jest generowany przez funkcje <code>open</code> , <code>read</code> lub <code>rename</code> .
ELOOP	<i>Too many levels of symbolic links.</i> (Zbyt dużo poziomów łączy symbolicznych). Jest zwracany, gdy system musi podążać za zbyt dużą liczbą łączy symbolicznych, kiedy próbuje znaleźć plik albo katalog. Może być generowany przez dowolną funkcję systemową, która ma parametr w postaci nazwy ścieżki.
EMFILE	<i>Too many open files in a process.</i> (Zbyt dużo otwartych plików w procesie). Zdarza się, kiedy otwierany jest plik; oznacza, że została osiągnięta granica otwartych deskryptorów pliku przypadających na proces. Granica ta jest określona przez <code>OPEN_MAX</code> w <code><limits.h></code> .
EMLINK	<i>Too many links.</i> (Zbyt dużo łączy). Generowany przez funkcję <code>link</code> , gdy została osiągnięta maksymalna liczba łączy związanych z pojedynczym plikiem fizycznym. Granica ta jest określona przez <code>LINK_MAX</code> w <code><limits.h></code> .
EMSGSIZE	<i>Message too large.</i> (Zbyt duży komunikat). Generowany w sieci, jeśli wysłany komunikat jest zbyt duży do zapamiętania w wewnętrznym buforze odbiorcy.
EMULTIHOP	<i>Reserved.</i> (Zarezerwowany).

ENAMETOOLONG	<i>Filename too long.</i> (Zbyt dłużna nazwa pliku). Może to oznaczać, że indywidualna nazwa pliku jest dłuższa niż <code>NAME_MAX</code> lub że nazwa ścieżki przekracza <code>PATH_MAX</code> . Może być zwrócony przez każdą funkcję systemową, która używa jako parametru nazwy ścieżki lub nazwy pliku.
ENETDOWN	<i>Network is down.</i> (Sieć nie działa).
ENETUNREACH	<i>Network unreachable.</i> (Sieć jest nieosiągalna). Nie jest dostępna żadna marszruta do osiągnięcia wymaganej sieci.
ENFILE	<i>File table overflow.</i> (Przepełnienie tablicy plików). Generowany przez funkcje, które zwracają deskryptor otwartego pliku (takie jak <code>creat</code> , <code>open</code> i <code>pipe</code>). Oznacza, że wewnętrzna tablica plików w jądrze jest pełna i nie może być otwartych więcej deskryptorów plików.
ENOBUFS	<i>No buffer space is available.</i> (Nie jest dostępna przestrzeń dla bufora). Ten błąd odnosi się do gniazda. Jeśli przestrzeń bufora nie jest dostępna dla dowolnej funkcji gniazda, zwarcany jest komunikat o błędzie.
ENODATA	<i>No message available.</i> (Nie jest dostępny komunikat). Zwarcany przez funkcję <code>read</code> , jeśli żaden komunikat nie czeka przy głowie STREAM.
ENODEV	<i>No such device.</i> (Nie ma takiego urządzenia). Zwarcany przy próbie wykonania niedozwolonej funkcji systemowej dla urządzenia (jak odczyt urządzenia tylko-do-zapisu).
ENOENT	<i>No such file or directory.</i> (Nie ma takiego pliku lub katalogu). Zdarza się, jeśli żaden rzeczywisty plik nie odpowiada nazwie ścieżki (na przykład przy funkcji <code>open</code>) albo jeden z katalogów w nazwie ścieżki nie istnieje.
ENOEXEC	<i>Exec format error.</i> (Błąd formatu wykonywalnego). Nie został rozpoznany prawidłowy format pliku programu wykonywalnego. Zwarcany przez funkcję <code>exec</code> .
ENOLCK	<i>No locks available.</i> (Blokady nie są dostępne). Nie ma dostępnych więcej blokad rekordów dla blokady <code>fcntl</code> .
ENOLINK	<i>Reserved.</i> (Zarezerwowany).
ENOMEM	<i>Not enough space.</i> (Nie ma wystarczającej pamięci). Ogólny błąd pamięci, zdarzający się, gdy proces prosi o więcej pamięci niż system może dostarczyć. Może być generowany przez funkcje <code>exec</code> i <code>fork</code> oraz procedury <code>brk</code> i <code>sbrk</code> , które są zainteresowane przydziałem pamięci.
ENOMSG	<i>No message of the desired type.</i> (Nie ma komunikatu żądanego typu). Zwarcany, jeśli <code>msgrcv</code> nie może znaleźć komunikatu albo żadanego typu komunikatu w kolejce komunikatów.

ENOPROTOOPT	<i>Protocol not available.</i> (Protokół nie jest dostępny). Żądany protokół nie jest dostępny dla funkcji systemowej gniazda.
ENOSPC	<i>No space left on device.</i> (Brak wolnej przestrzeni w urządzeniu). Urządzenie jest pełne i nie może być ani rozszerzony plik, ani utworzona nowa pozycja katalogu. Może być generowany przez funkcje write, creat, open, mknod i link.
ENOSR	<i>No stream resources.</i> (Brak zasobów strumienia). Stan tymczasowy, relacjonowany, gdy zasób pamięci dla strumieni STREAM nie jest dostępny.
ENOSTR	<i>Not a STREAM.</i> (To nie jest STREAM). Zwracany, jeśli funkcja strumienia, taka jak ioctl dla umieszczenia na stosie jest wywoływana dla urządzenia nie-strumieniowego.
ENOSYS	<i>Function not implemented.</i> (Funkcja nie zaimplementowana). Oznacza to, że zostało wykonane wywołanie funkcji systemowej, która nie jest dostępna w bieżącej implementacji.
ENOTCONN	<i>Socket not connected.</i> (Gniazdo nie połączone). Ten błąd jest generowany, jeśli wywoływane są funkcje sendmsg albo recvmsg w odniesieniu do gniazda, które nie ma żadnego połączenia.
ENOTDIR	<i>Not a directory.</i> (To nie jest katalog). Zdarza się, jeśli nazwa ścieżki nie reprezentuje katalogu, gdy kontekst tego żąda. Może być ustawiany przez funkcje chdir, mkdir, link i wiele innych.
ENOTEMPTY	<i>Directory not empty.</i> (Katalog nie jest pusty). Na przykład zwracany przez rmdir, jeśli wykonano próbę usunięcia katalogu, który nie jest pusty.
ENOTSOCK	<i>Not a socket.</i> (To nie jest gniazdo). Deskryptor pliku użyty przez funkcję sieciową, taką jak connect, nie jest deskryptorem gniazda.
ENOTTY	<i>Not a character device.</i> (To nie jest urządzenie znakowe). Zostało wykonane wywołanie ioctl w odniesieniu do otwartego pliku, który nie jest specjalnym urządzeniem znakowym.
ENXIO	<i>No such device or address.</i> (Nie ma takiego urządzenia lub adresu). Zdarza się, kiedy wykonano próbę dostępu do urządzenia lub adresu urządzenia, które nie istnieją. Błąd ten mogą powodować urządzenia odłączone.
EOPNOTSUPP	<i>Operation not supported on a socket.</i> (Operacja nie obsługiwana przez gniazdo). Rodzina adresów związana z gniazdem nie obsługuje wywołania funkcji.
EOVERFLOW	<i>Value too large to be stored in the data type.</i> (Wartość zbyt duża, aby ją zapamiętać w danym typie danej).
EPERM	<i>Operation not permitted.</i> (Operacja nie dozwolona). Wskazuje, że proces próbuje manipulować plikiem w sposób dozwolony

EPPIPE	tylko właścielowi pliku lub super-użytkownikowi (root). Może też oznaczać, że próbowano wykonać operację dozwoloną tylko dla super-użytkownika.
EPROTO	<i>Broken pipe.</i> (Potok uszkodzony). Ustawiany przez funkcję write; oznacza, że była wykonana próba zapisu do potoku, który nie jest otwarty do odczytu przez żaden proces; w rzeczywistości ten stan zwykle powoduje przerwanie procesu zapisu przez sygnał SIGPIPE. Błąd EPIPE jest ustawiany tylko wtedy, jeśli sygnał SIGPIPE został przechwycony i proces się nie zakończył.
EPROTONOSUPPORT	<i>Protocol error.</i> (Błąd protokolu). Ten błąd jest specyficzny dla urządzenia i wskazuje, że został odebrany błąd protokolu.
EPROTOTYPE	<i>Protocol not supported.</i> (Protokół nie obsługiwany). Jest zwracany przez funkcję systemową gniazda, jeśli rodzina adresów nie jest obsługiwana przez system.
ERANGE	<i>Socket type not supported.</i> (Ten typ gniazda nie jest obsługiwany). Jest także zwracany przez funkcję gniazda, jeśli taki typ protokolu jak SOCK_DGRAM, nie jest obsługiwany przez system.
EROFS	<i>Result too large or too small.</i> (Wynik zbyt wielki albo zbyt mały). Błąd matematyczny oznaczający, że zwracana wartość funkcji nie może być reprezentowana na procesorze hosta.
ESPIPE	<i>Read-only file system.</i> (System plików tylko-do-odczytu). W systemie plików, który został dołączony w celach bezpieczeństwa jako tylko-do-odczytu, była wykonana próba zapisu albo modyfikacji pozycji katalogu.
ESRCH	<i>Illegal seek.</i> (Nielegalne szukanie). Było wykonane nie mające znaczenia wywołanie funkcji lseek w odniesieniu do potoku.
ESTALE	<i>No such process.</i> (Nie ma takiego procesu). Wyszczególniony został nie istniejący proces. Generowany przez kill.
ETIME	<i>Reserved.</i> (Zarezerwowany).
ETIMEDOUT	<i>ioctl timeout on a STREAM.</i> (Upłynął czas oczekiwania ioctl dla STREAM). Przy wywołaniu ioctl dla STREAM może być ustawiony czas oczekiwania, który właśnie upłynął. Może to wskazywać, że czas oczekiwania powinien być wydłużony.
ETXTBSY	<i>Connection timed out.</i> (Upłynął czas oczekiwania na połączenie). Kiedy proces próbuje połączyć się z innym systemem, czas oczekiwania może zostać przekroczony, jeśli system nie działa lub jest zbyt dużo żądań czekających na to konkretne połączenie.
	<i>Text file busy.</i> (Plik tekstowy jest zajęty). Jeśli jest generowany przez funkcję exec, to znaczy, że wykonana była próba uruchomienia programu ze wspólnym tekstem, który jest aktual-

EWOULDBLOCK

nie otwarty do zapisu. Jeśli jest generowany przez funkcję, która zwraca deskryptory pliku, to znaczy, że była wykonana próba otwarcia do zapisu programu ze wspólnym tekstem, który jest wykonywany.

Operation would block. (Operacja może się zablokować). Ten błąd będzie zwrócony, jeśli gniazdo zostało otwarte jako nie blokujące, a operacja taka jak odczyt lub zapis mogłaby je zablokować. Błąd EWOULDBLOCK powinien mieć tę samą wartość, co EAGAIN w systemie zgodnym z XSI.

EXDEV

Cross-device link. (Łącze między urządzeniami). Zdarza się, jeśli wywoływana jest funkcja link do łączenia plików w różnych systemach plików.

DODATEK B

Główne standardy

B.1 Historia

Historia Uniksa jest dłuża i sięga czasów, gdy Ken Thomson, Dennis Ritchie i inni zaczynali pracę na mało używanym minikomputerze PDP-7 w kąciku sali w roku 1969. Od tego czasu Unix pojawił się w wielu wcieleniach, jako wersja handlowa, wariant akademicki czy też podstawa wysiłków normalizacyjnych.

Oto przegląd wybranych ważniejszych wersji lub wydarzeń:

- *Sixth Edition* lub *Version 6* (1975). Pierwsza szeroko dostępna wersja, przyjmniej w społeczności technicznej, i podstawa dla pierwszego Uniksa Berkeley.
- *Xenix* (1980). Wersja Microsoftu – jedna z dziwnie nazwanych wersji handlowych z wczesnych lat osiemdziesiątych.
- *System V* (1983-1992). Jedna z najbardziej ekspansywnych wersji Uniksa z AT&T, twórcy systemu Unix. Potomek Version 7 i późniejszego System III.
- *Berkeley UNIX* (wersja 4.2 w 1984 roku, 4.4 w 1993 roku). Została opracowana na Uniwersytecie Berkeley; jedna z najważniejszych wersji Uniksa, z wieloma dodatkowymi cechami.
- *POSIX* (1988 i dalej). Początek zestawu standardów z IEEE (patrz poniżej).
- *X/Open Portability Guides* (XPG3 w 1990 roku, XPG4.2 w 1994 roku). Specyfikacja praktyczna, jednocześnie pewną liczbę podstawowych standardów i praktyki przemysłowej. X/Open ostatecznie zdobył znak towarowy Uniksa.

Miało też miejsce wiele transakcji handlowych, takich jak przeniesienie znaku UNIX od AT&T najpierw do Novella, a następnie do X/Open, później zaś połączenie X/Open i Open Software Foundation. To uproszczony obraz przeszłości, prawdziwe drzewo genealogiczne jest znacznie bardziej złożone.

B.2 Kluczowe standardy

Oto bieżące standardy, istotne dla treści tej książki:

SVID

SVID oznacza *System V Interface Definition* firmy AT&T. Został pierwotnie opracowany na wiosnę 1985 roku w celu objaśnienia standardowego interfejsu dla Unixa w wersji System V. *SVID* miał pewną liczbę wydań, kończąc się na trzecim wydaniu w 1989 roku. Pierwsze wydanie tej książki było oparte na *SVID*.

ANSI C

Komitet ANSI określa standardy dla wszystkich kwestii związanych z programowaniem. Szczególnie interesującym standardem dla programistów systemu Unix jest ANSI C.

IEEE/POSIX

Institute of Electrical and Electronics Engineers (IEEE) opracował, wśród innych rzeczy, standard dla *Portable Operating Systems Interface (POSIX)*, który wywodzi się bezpośrednio z Unixa. Ten standard został po raz pierwszy opublikowany w 1988 roku i miał kilka aktualizacji. Dwie najbardziej istotne dla treści tej książki to:

1. IEEE Std 1003.1-1990, identyczny z ISO POSIX-1-ISO/IEC 9945-1:1990, Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language].
2. IEEE Std 1003.2-1992, identyczny z ISO POSIX-2-ISO/IEC 9945-2:1993, Information Technology – Portable Operating System Interface (POSIX) – Part2: Shell and Utilities.

Rozszerzenia i dodatki: 1003.1b-1993, 1003.1c-1995, 1003.1i-1995 zawierają takie tematy, jak cechy czasu rzeczywistego i wątki.

X/Open (teraz Open Group)

X/Open Group łączy wspominane wyżej standardy, i inne, w jedną całość o nazwie Common Applications Environment (CAE) Specification. CAE zawiera interfejs systemowy i interfejs sieciowy.

Od czasu pierwszej publikacji w lipcu 1985 roku powstała też pewna liczba korekt. Treści tej książki najbardziej dotyczy *Issue 4 Version 2 X/Open CAE Specification*, z sierpnia 1994 roku, zawierająca interfejs systemowy i nagłówki. Stanowi to stałą bazę, ale zauważ, że *Issue 5 CAE Specification* (z 1997 roku) zawiera kilka z nowszych własności POSIX-a dotyczących czasu rzeczywistego i wątków oraz inne rozszerzenia z praktyki przemysłowej. Ta aktualizacja jest także zawarta w *Version 2 of the Single UNIX Specification* Open Group i odmianie *UNIX98*.

BIBLIOGRAFIA

- Bach, M.J.: *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
- Curry, D.A.: *UNIX System Programming for SVR4*, O'Reilly & Associates, Inc., Sebastopol, CA, 1996.
- Dijkstra, E.W.: 'Co-operating Sequential Processes' w Genugs, F. (wyd.): *Programming Languages*, Academic Press, 1968.
- Galimeister, B.O.: *POSIX.4: Programming for the Real World*, O'Reilly & Associates, Inc., Sebastopol, CA, 1995.
- Kernighan, B.W., Pike, R.: *The UNIX Programming Environment*, Prentice-Hall, 1984.
- Kernighan, B.W., Ritchie, D.: *The C Programming Language*, Prentice-Hall, 1978.
- Northrup, C.J.: *Programming with UNIX Threads*, Wiley, New York, NY, 1996.
- Stevens, W.R.: *Advanced Programming in the UNIX Environment*, Addison-Wesley, Reading, MA, 1992.
- Pr. zbiorowa: *The Bell System Technical Journal (Computer Science and Systems)*, AT&T Bell Laboratories, lipiec – sierpień 1978.
- Pr. zbiorowa: *AT&T Bell Laboratories Technical Journal (Computer Science and Systems)*, *The UNIX System*, AT&T Bell Laboratories, lipiec – sierpień 1984.

INDEX

A

abs, 313
accept, 249
access, 44
adres
 internetowy, 245
 sieciowy, 245
AF_INET wartość, 247
AF_UNIX wartość, 247
alarm, 138
anormalne zakończenie procesu, 124
argumenty wywołania programu, 90
arpa/inet.h plik nagłówkowy, 245
asctime, 306
atexit, 97
atof, 309
atoi, 25, 309
atol, 309
atrybuty procesu, 109

B

bezpośredni dostęp do pliku, 21, 276
bezwzględna nazwa ścieżki, 2
biała spacja, 283
bieżący katalog
 główny, 115
 roboczy, 3, 59, 69, 115
bind, 248
bit klejący, 41
blok dyskowy, 17
blokada
 odczytu, 173
 zapisu, 173
blokowanie
 rekordu, 171, 173
 sygnalu, 134
brk, 313
buforowanie w pamięci podręcznej, 75
BUFSIZ stała, 17

C

c_cc tablica, 224
calloc, 296
catopen, 313
cbrt, 313
cfgetispeed, 225

cfgetospeed, 225
cfsetispeed, 225
cfsetospeed, 225
chdir, 70
chmod, 46, 55
chown, 46
clearerr, 270
close, 7, 14
closedir, 66
close-on-exec znacznik, 96
connect, 251
creat, 7, 13
ctime, 192
ctype.h plik nagłówkowy, 310
cuserid, 312
czas oczekiwania, 217
czasomierz, 121
czytanie katalogu, 66

D

datagram, 244
#define, 10
demon, 112
deskryptor pliku, 8
 dziedziczenie, 94
difftime, 306
dirent struktura, 64
dirent.h plik nagłówkowy, 64, 66
div, 313
dodawanie danych do pliku, 25
drand48, 313
drzewo katalogów, 71
dwukierunkowa lista powiązana, 298
dynamiczny przydział pamięci, 295
dziedziczenie
 danych, 94
deskryptora pliku, 94

E

E2BIG, 316
EACCES, 43, 316
EADDRINUSE, 316
EADDRNOTAVAIL, 316
EAFNOSUPPORT, 316
EAGAIN, 154, 316
EALREADY, 316

EBADFE, 316
 EBADMSG, 316
 EBUSY, 317
 ECHILD, 99, 317
 ECHO znacznik, 227
 ECHOE znacznik, 227
 ECHOK znacznik, 227
 ECHONL znacznik, 227
 ECONNABORTED, 317
 ECONNREFUSED, 317
 ECONNRESET, 317
 EDEADLK, 317
 EDESTADDRREQ, 317
 EDOM, 317
 EDQUOT, 317
 EEXIST, 44, 317
 EFAULT, 317
 EFBIG, 317
 efektywność odczytu-zapisu, 20
 efektywny

- identyfikator grupy, 38, 116
- identyfikator użytkownika, 38
- user-id, 41

 ejid, 38
 EHSTUNREACH, 317
 EIDRM, 317
 EILSEQ, 318
 EINPROGRESS, 318
 EINTR, 318
 EINVAL, 318
 EIO, 318
 EISCONN, 318
 EISDIR, 60, 318
 ELOOP, 318
 EMFILE, 146, 318
 EMLINK, 318
 EMSGSIZE, 318
 EMULTIHOP, 319
 ENAMETOOLONG, 319
 ENETDOWN, 319
 ENETUNREACH, 319
 ENFILE, 319
 ENOBUFFS, 319
 ENODATA, 319
 ENODEV, 319
 ENOENT, 319
 ENOEXEC, 319
 ENOLCK, 319
 ENOLINK, 319
 ENOMEM, 319
 ENOMSG, 319
 ENOPROTOOPT, 320
 ENOSPC, 320
 ENOSR, 320
 ENOSTR, 320
 ENOSYS, 320

ENOTCONN, 320
 ENOTDIR, 320
 ENOTEMPTY, 320
 ENOTSOCK, 320
 ENOTTY, 320
 ENXIO, 320
 EOF stala, 33
 EOPNOTSUPP, 320
 EVERFLOW, 320
 EPERM, 320
 EPIPE, 154, 321
 EPROTO, 321
 EPROTONOSUPPORT, 321
 EPROTOTYPE, 321
 ERANGE, 321
 erff, 313
 EROFS, 321
 errno zmienna błędu, 34, 315
 errno.h plik nagłówkowy, 35, 315
 ESPIPE, 321
 ESRCH, 321
 ESTALE, 321
 ETIME, 321
 ETIMEDOUT, 321
 ETXTBSY, 321
 euid, 38
 EWOULDBLOCK, 322
 EXDEV, 322
 exec1, 89
 exec1p, 89
 execv, 89
 execvp, 88, 89
 exit, 11, 97

F

F_GETFL wartość, 27
 F_GETLK wartość, 174, 178
 F_OK wartość, 45
 F_RDLCK wartość, 175
 F_SETFL wartość, 27
 F_SETLK wartość, 174, 178
 F_SETLKW wartość, 174
 F_UNLCK wartość, 175, 177
 F_WRLCK wartość, 175
 fclose, 262
 fcntl, 7, 26, 154, 173, 174
 fcntl.h plik nagłówkowy, 8, 10, 174
 FD_CLR makroinstrukcja, 158
 FD_ISSET makroinstrukcja, 157
 FD_SET makroinstrukcja, 157
 fd_set typ, 158
 FD_SETSIZE wartość, 157
 FD_ZERO makroinstrukcja, 157
 fopen, 292
 feof, 270
 ferror, 270

Indeks

fflush, 264
 fgets, 271
 FIFO, 147
 FILE struktura, 32, 261
 fileno, 270
 flock struktura, 174
 floor, 313
 fopen, 32, 262
 fork, 84
 formatowane wyjście, 277
 fpathconf, 80
 fprintf, 34, 277
 fputs, 273
 fread, 273
 freopen, 292
 frexp, 313
 fscanf, 283
 fseek, 276
 fstat, 50, 154,
 fstatvfs, 79
 fsync, 75
 ftell, 276
 ftok, 182
 ftw, 71
 ftw.h plik nagłówkowy, 72
 FTW_D wartość, 72
 FTW_DNR wartość, 72
 FTW_F wartość, 72
 FTW_NS wartość, 72
 FTW_SL wartość, 72
 funkcja systemowa, 4

- a podprogram, 5

 fwrite, 273

G

gamma, 313
 get, 182
 GETALL wartość, 197
 getc, 33, 265
 getch, 104
 getcwd, 70
 getenv, 114
 getgrant, 312
 getlogin, 312
 GETNCNT wartość, 197
 getpgrp, 111
 GETPID wartość, 197
 getpwent, 312
 getrlimit, 312
 gets, 271
 getsid, 112
 GETVAL wartość, 197
 GETZCNT wartość, 197
 gid_t typ, 46
 główny numer urządzenia, 78
 gmtime, 306

gniazdo, 4, 243, 312
 ogólne, 246
 zamknięcie, 254

grupa

- pliku, 38
- procesów, 111

H

hypot, 313

I

I/O odwzorowane w pamięci, 301
 ICANON znacznik, 227
 ICRNL znacznik, 226
identyfikator

- grupy procesów, 111
- procesu, 86
- sesji, 112
- urządzenia IPC, 182
- użytkownika, 37
- zestawu semaforów, 195

 TEXTEN znacznik, 227
 IGNCR znacznik, 226
 ignorowanie sygnału, 130
 implementacja katalogu, 60
 in_addr_t typ, 245
 inet_addr, 245
 informacja o systemie plików, 79
 init, 84
 INLCR znacznik, 226
 interfejs funkcji systemowych, 4
 interfejs gniazd, 246
 _IOBNF wartość, 293
 _IOMBF wartość, 293
 _IOLBF wartość, 293
 IPC_CREAT wartość, 184
 IPC_EXCL wartość, 184
 IPC_NOWAIT wartość, 185
 IPC_RMID wartość, 192, 196
 IPC_SET wartość, 192, 196
 IPC_STAT wartość, 192, 196
 ipcrn, 208
 ipcs, 208
 isalnum, 313
 isalpha, 310
 isascii, 310
 isatty, 221
 iscntrl, 310
 isdigit, 310
 isgraph, 310
 ISIG znacznik, 227
 islower, 310
 isprint, 310
 ispunct, 310
 isspace, 310
 isupper, 310

isxdigit, 310
i-węzły, 51, 60
IXANY znacznik, 226
IXOFF znacznik, 226
IXON znacznik, 226

J

jądro, 4
jednokierunkowa lista powiązana, 298

K

kasowanie pliku, 26
katalog
 bieżący główny, 115
 bieżący roboczy, 59, 115
 czytanie, 66
 główny, 2, 57, 58
 implementacja, 60
 kropka, 62
 macierzysty, 43, 58
 ograniczenia systemowe, 80
 otwieranie, 65
 podwójna kropka, 62
 prawa dostępu, 63
 roboczy, 3
 tworzenie, 64
usuwanie, 64
zamykanie, 65
zmiana, 70
key_t typ, 181
kill, 120, 136
klucz urządzenia IPC, 181
kolejka
 komunikatów, 183
 priorytetowa, 186
komunikacja międzymiędzynarodowa, 4, 171
komunikat
 rozgłoszeniowy, 244
kończenie procesu, 97
kropka w katalogu, 62

L

leksem, 104
liczba magiczna, 88
licznik łączy, 47
link, 47, 62
lista powiązana, 297
listen, 249
lockf, 173
log, 313
ls, 165
lseek, 7, 21

L

ładowanie początkowe, 62
łącze
 symboliczne, 49
 twarde, 47

M

macierzysty katalog, 437
malloc, 295
MAP_PRIVATE wartość, 303
MAP_SHARED wartość, 303
maska tworzenia pliku, 42
math.h plik nagłówkowy, 313
memchr, 301
memcmp, 301
memcpy, 301
memmove, 301
memset, 301
MIN parametr, 229
mkdir, 65
mknod, 165
mktimes, 306
mmap, 302
mode_t typ, 42
model
 bezpośredni, 244, 256
 połączeniowy, 244, 247
moduł protokołów, 210
MSG_DONTROUTE wartość, 252
MSG_NOERROR wartość, 186
MSG_OOB wartość, 252
MSG_PEEK wartość, 252
MSG_WAITALL wartość, 252
msgctl, 182, 191
msgget, 182, 183
msgrecv, 182, 185
msgsnd, 182, 185
msqid_ds struktura, 191
munmap, 303

N

nazwa
 pliku, 2
 ścieżki, 2
 użytkownika, 43
nazwany potok, 165
nice, 117
niezmiennik semafora, 194
NOFLSH znacznik, 227
NULL wskaźnik, 32
numer i-węzła, 60
numer portu, 246

O

O_ACCMODE znacznik, 27
O_APPEND znacznik, 18, 25
O_EXCL znacznik, 13
O_NDELAY znacznik, 154
O_NONBLOCK znacznik, 154
O_RDONLY stała, 8
O_RDWR znacznik, 10
O_TRUNC znacznik, 13
O_WRONLY znacznik, 10
obwód wirtualny, 244
OCRNL znacznik, 227
odłączalny wolumen, 74
odstęp, 283
off_t typ, 22
ograniczenie
 liczby otwartych plików, 11
 systemowe, 80
 wielkość pliku, 117
ONLCR znacznik, 227
ONLRET znacznik, 227
open, 7, 10, 43, 217
opendir, 65
operacja niepodzielna, 153
otwieranie katalogu, 65

P

PAREN znacznik, 226
PARODD znacznik, 226
parzystość, 226
pathconf, 80
pause, 140
pclose, 288
perror procedura, 35
perror, 315
pid, 86
pid_t typ, 85
pipe, 146
plik
 bezpośredni dostęp, 21, 276
 deskryptory, 8
 dodawanie danych, 25
 efektywność odczytu-zapisu, 20
 FIFO, 165
 grupa, 43
 kasowanie, 26
 maska tworzenia, 42
 nazwa, 2
 obcięcie, 13
 ograniczenia systemowe, 80
 operacja elementarna, 7
 prawa dostępu, 12, 38
 prawidłowy, 3
 specjalny, 57
strumień, 262
tryb otwarcia, 10
tryb, 39
tylko do odczytu, 8
uogólniony, 3
uprawnienia przy tworzeniu, 43
uprawnienia, 38, 43
urządzenia blokowego, 77
urządzenia znakowego, 77
urządzenia, 76
wiele nazw, 47
właściciel, 37
wskaźnik zapisu-odczytu, 16
zmiana uprawnień, 45
zmiana właściciela, 46
zwykły, 3
poboczny numer urządzenia, 78
podkatalog, 58
podprogram, 5
 a funkcja systemowa, 5
podwójna kropka w katalogu, 62
popen, 288
port, 246
potok, 4, 84, 145
 na poziomie powłoki, 162
 nazwany, 165
 wielkość, 151
 zamykanie, 153
potomek, 85
powłoka, 4
prawa dostępu, 12, 38
 dla katalogu, 63
prawidłowy plik, 3
prawo przeszukiwania, 64
printf, 34, 277
priorytet procesu, 117
proces, 4, 83
 anormalne zakończenie, 124
 atrybuty, 109
 kończenie, 109
 potomny, 85
 priorytet, 117
 rodzicielski, 85
 synchronizacja, 99
 zmiana grupy, 111
 zombie, 102
procesor poleceń, 103
process_id, 86
PROT_EXEC wartość, 303
PROT_NONE wartość, 303
PROT_READ wartość, 303
PROT_WRITE wartość, 303
przechodzenie drzewa katalogów, 71
przechwytywanie sygnału, 128
przekierowanie, 28

przelącznik
 urządzeń blokowych, 77
 urządzeń znakowych, 77
przerwanie, 119
pseudoterminal, 232
punkt końcowy transportu, 246
putc, 33, 265
puts, 273

R

R_OK wartość, 45
raise, 138
read, 7, 15, 218
 efektywność, 20
readdir, 66
readlink, 50
realloc, 297
recv, 252
recvfrom, 257
relacja własności, 3
remove, 7, 26
rename, 49
rewind, 276
rewinddir, 67
rmdir, 65
rodzic, 85
rozszerzenie czasu rzeczywistego, 312
ruid, 38
rzeczywisty identyfikator
 grupy, 116
 użytkownika, 38, 116

S

S_IFBLK wartość, 78
S_IFCHR wartość, 78
S_IGRP wartość, 39
S_IROTH wartość, 39
S_IRUSR wartość, 39
S_ISGID wartość, 43
S_ISUID wartość, 43
S_ISVTX wartość, 40
S_IWGRP wartość, 39
S_IWOTH wartość, 39
S_IWUSR wartość, 39
S_IXGRP wartość, 39
S_IXOTH wartość, 39
S_IXUSR wartość, 39
sbrk, 301
scanf, 283
ścieżka, 2
 nazwa bezwzględna, 2
 nazwa względna, 3
SEEK_CUR znacznik, 22
SEEK_END znacznik, 22
SEEK_SET znacznik, 22

sekcja krytyczna, 195
sekwencja unikowa, 209
select, 157
SEM_UNDO znacznik, 200
semafor, 4, 194
 niezmiennik, 194
semctl, 196
semget, 195
semid_ds struktura, 196
semop, 198
send, 252
sendto, 257
sesja, 112
session-id, 112
SETALL wartość, 197
setbuf, 292
setgid, 116
setlocale, 313
setpgid, 111
setsid, 112
setuid, 116
SETVAL wartość, 197
setvbuf, 292
SHM_RDONLY znacznik, 203
SHM_RND znacznik, 203
shmatt, 202
shmat, 203
shmemt, 203
shmget, 202
SIG_DFL wartość, 128
SIG_IGN wartość, 128
SIGABRT sygnał, 121
sigaction struktura, 127
sigaction, 127
sigaddset, 126
SIGALRM sygnał, 121
SIGBUS sygnał, 121
SIGCHLD sygnał, 121
SIGCONT sygnał, 121
sigdelset, 126
sigemptyset, 126
SIGEMT sygnał, 124
sigfillset, 126
SIGFPE sygnał, 121
SIGHUP sygnał, 121, 231
SIGILL sygnał, 122
SIGINT sygnał, 122
SIGKILL sygnał, 122
siglongjmp, 133
signal.h plik nagłówkowy, 126
SIGPIPE sygnał, 122
SIGPOLL sygnał, 122
sigprocmask, 134
SIGPROF sygnał, 122
SIGQUIT sygnał, 122
sigset_t typ, 126
sigsetjmp, 133

SIGSTOP sygnał, 123
SIGTERM sygnał, 123
SIGTRAP sygnał, 123
SIGTSTP sygnał, 123
SIGTTIN sygnał, 123
SIGTTOU sygnał, 123
SIGURG sygnał, 123
SIGUSR1 sygnał, 123
SIGUSR2 sygnał, 123
SIGVTALRM sygnał, 123
SIGXCPU sygnał, 124
SIGXFSZ sygnał, 124
sinh, 313
size_t typ, 15
sleep, 31
SOCK_DGRAM wartość, 247
SOCK_STREAM wartość, 247
sockaddr struktura, 246
socket, 246
specyfikator konwersji, 277
sprintf, 277, 282
sqrt, 313
środowisko, 113
sscanf, 283
ssize_t typ, 9
st_mode, 78
st_rdev, 78
stan wyjścia, 11
Standardowa Biblioteka I/O, 5, 261
 przegląd, 32
standardowe wejście, 28
standardowe wyjście komunikatów o błędach, 28
standardowe wyjście, 28
stat struktura, 50, 51, 78, 154
statvfs struktura, 79
statvfs, 79
stderr, 34, 157, 268
stdin, 157, 268
stdio.h plik nagłówkowy, 32, 262
stdlib.h plik nagłówkowy, 11, 286, 295
stdout, 157, 268
sterownik
 terminala, 209
 urządzenia, 210
strcasecmp, 307
strcat, 307
strchr, 307
strcmp, 307
strcpy, 307
strcspn, 307
strupr, 307
STREAM, 234
string.h plik nagłówkowy, 301, 307
strlen, 307

strncasecmp, 307
strncat, 307
strncmp, 307
strncpy, 307
strpbrk, 307
strrchr, 307
strspn, 307
strstr, 307
strtod, 309
strtok, 307
strtol, 309
struktura stanu urządzenia IPC, 183
strumień pliku, 262
super-blok, 74
sygnał, 4, 100
 blokowanie, 134
 ignorowanie, 130
 obsługa, 125
 przechwytywanie, 128
 wysyłanie, 136
 zawieszenia, 231
 zestaw, 126
symlink, 50
sync, 75
synchronizacja procesów, 99
sys/mman.h plik nagłówkowy, 302
sys/msg.h plik nagłówkowy, 191
sys/sem.h plik nagłówkowy, 196
sys/shm.h plik nagłówkowy, 202
sys/socket.h plik nagłówkowy, 246
sys/stat.h plik nagłówkowy, 39, 43, 51
sys/statvfs.h plik nagłówkowy, 79
sys/time.h plik nagłówkowy, 157
sys/types.h plik nagłówkowy, 9, 22, 43
sys/wait.h plik nagłówkowy, 101
sys_errlist tablica, 315
sys_nerr wartość, 315
sysconf, 312
system plików, 49, 57, 73
 informacja, 79
system, 286

T

tcdrain, 230
tcflow, 230
tflush, 230
tcgetattr, 222
TCSADRAIN wartość, 222
TCSAFLUSH wartość, 222
TCSANOW wartość, 222
tcsendbrk, 230
tcsetattr, 222
terminal sterujący, 112, 213
termios struktura, 221, 223
termios.h plik nagłówkowy, 222

TIME parametr, 229
 time, 305
 time.h plik nagłówkowy, 305
 time_t typ, 305, 312
 timeval struktura, 159
 tm struktura, 306
 toascii, 311
 TOSTOP znacznik, 227
 toupper, 311
 trig, 313
 tryb pliku, 39
 tryb otwarcia
 stała numeryczna, 12
 kanoniczny, 214
 pełnego dupleksu, 214
 ttyname, 221
 tworzenie katalogu, 64

U

uid, 37
 uid_t typ, 46
 ulimit, 117
 umask, 42
 uname, 312
 ungetc, 267
 unistd.h plik nagłówkowy, 9, 21, 32
 unlink, 7, 26, 48, 61
 uogólniony plik, 3
 uprawnienia, 3, 38
 urządzenie
 blokowe, 77
 IPC, 171
 IPC, identyfikator, 182
 IPC, struktura stanu, 183
 klucz, 181
 nadrgzdne, 232
 numer główny, 78
 numer poboczny, 78
 pierwotne, 77
 podrzędne, 232
 znakowe, 77
 user-id, 37
 usuwanie katalogu, 64

V

VEOF parametr, 224
 VEOL parametr, 224
 VERASE parametr, 224
 VINTR parametr, 224
 VKILL parametr, 224
 VMIN parametr, 229
 VQUIT parametr, 224
 VSTART parametr, 224

VSTOP parametr, 224
 VSUSP parametr, 224
 VTIME parametr, 229

W

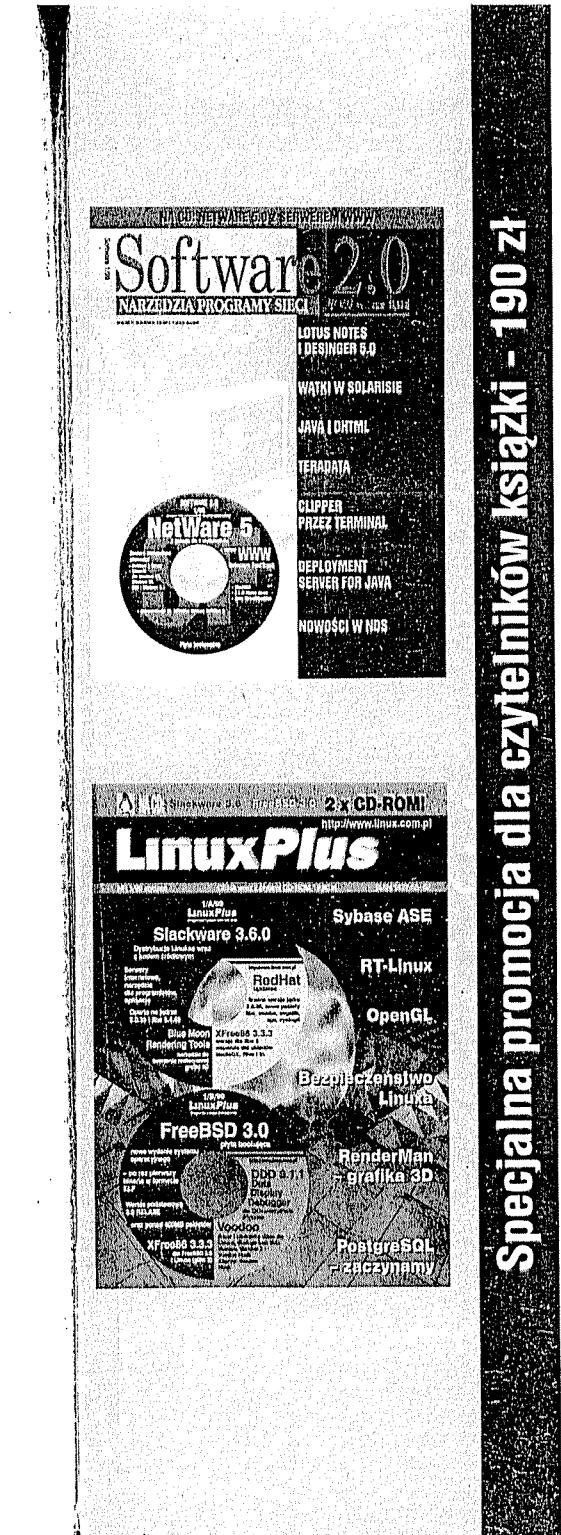
W_OK wartość, 45
 wait, 92, 99
 waitpid, 101
 wątek, 83, 312
 WEXITSTATUS makroinstrukcja, 100
 wiele nazw pliku, 47
 wielkość potoku, 151
 WIFEXITED makroinstrukcja, 100
 właściciel pliku, 37
 własność, 3
 WNOHANG wartość, 101
 wolumen odlączalny, 74
 write, 7, 18, 220
 efektywność, 20
 wskaźnik
 pliku, 16
 pusty, 66
 zapisu-odczytu, 16
 wspólna pamięć, 4, 202
 wysyłanie sygnału, 136
 wzajemne wykluczanie, 194
 względna nazwa ścieżki, 3

X

X_OK wartość, 45

Z

zadanie, 83
 zakleszczenie, 179
 zakończenie anormalne, 121
 zamknięcie
 połączenia, 254
 katalogu, 65
 zegar alarmu, 121
 zestaw sygnałów, 126
 zmiana
 grupy procesu, 111
 katalogu, 70
 uprawnień pliku, 45
 właściciela pliku, 46
 zombie proces, 102
 zrzut
 rdzenia, 121
 zawartości pamięci, 125
 związywanie, 248
 zwykły plik, 3



Specjalna promocja dla czytelników książek - 190 zł

WPŁATA NA ROCZNĄ PRENUMERATĘ

<input type="checkbox"/> Linux Plus (+CD)	190 zł	od numeru.....
<input type="checkbox"/> Software (+CD)	190 zł	od numeru.....

WPŁATA NA ROCZNĄ PRENUMERATĘ

<input type="checkbox"/> Linux Plus (+CD)	190 zł	od numeru.....
<input type="checkbox"/> Software (+CD)	190 zł	od numeru.....

WPŁATA NA ROCZNĄ PRENUMERATĘ

<input type="checkbox"/> Linux Plus (+CD)	190 zł	od numeru.....
<input type="checkbox"/> Software (+CD)	190 zł	od numeru.....

ANKIETA

1. Podstawowa działalność 2. Stanowisko:

- moje firmy:
 Prezes
 Dyrektor
 Kierownik
 Gd. Informatyk
 Konsultant
 Administrator
 Usług
 Handel
 Uslugi
 Bankowość/Finanse
 Wisko/Polityka
 Edukacja
 Doradztwo/Konsulting
 Transport
 Inne.....
3. Jaki programistów pracuje w + ilu informacyjnych przejęte firmach:
 a)
 b)
 c)
4. Jaki czaszę o charakterze informacyjnym w firmach w którym:

