Łącza nienazwane

Paweł Garbacki

18 grudnia 2000

1 Co to są łacza nienazwane

Łacza nienazwane (ang. *pipes*) są jednym z podstawowych mechanizmów komunikacji miedzyprocesowej. Umożliwiają one procesom przesyłanie sobie nawzajem informacji podzielonych na niewielkie pakiety. Zapisywanie i odczytywanie danych odbywa się na zasadzie kolejki **FIFO** (ang. *first in first out*), czyli najpierw czytane są bajty, które przyszły najwcześniej. Używanie łączy ma sens jeśli przepływ informacji jest w danej chwili jednostronny w ramach jednego procesu, tzn. unikamy sytuacji w której jeden proces zapisuje dane do łącza, a zaraz potem je pobiera. Implementacja łączy nie zabrania takiego sposobu ich wykorzystania, ale nie ma on wielu sensownych zastosowań. Łatwość eksloatacji została okupiona dość istotnymi ograniczeniami funkcjonalnymi. Najważniejszym z nich jest wymóg, aby przesyłające sobie komunikaty procesy były ze sobą spokrewnione.

Uwaga W poniższym tekście używajac nazwy *łącza* mam na myśli *łącza nienazwane*. Mimo tej konwencji wiele ze stwierdzeń jest prawdziwych także dla **łączy nazwanych**, które są zaimplementowane na bazie właśnie łączy nienazwanych.

2 Podstawowe struktury

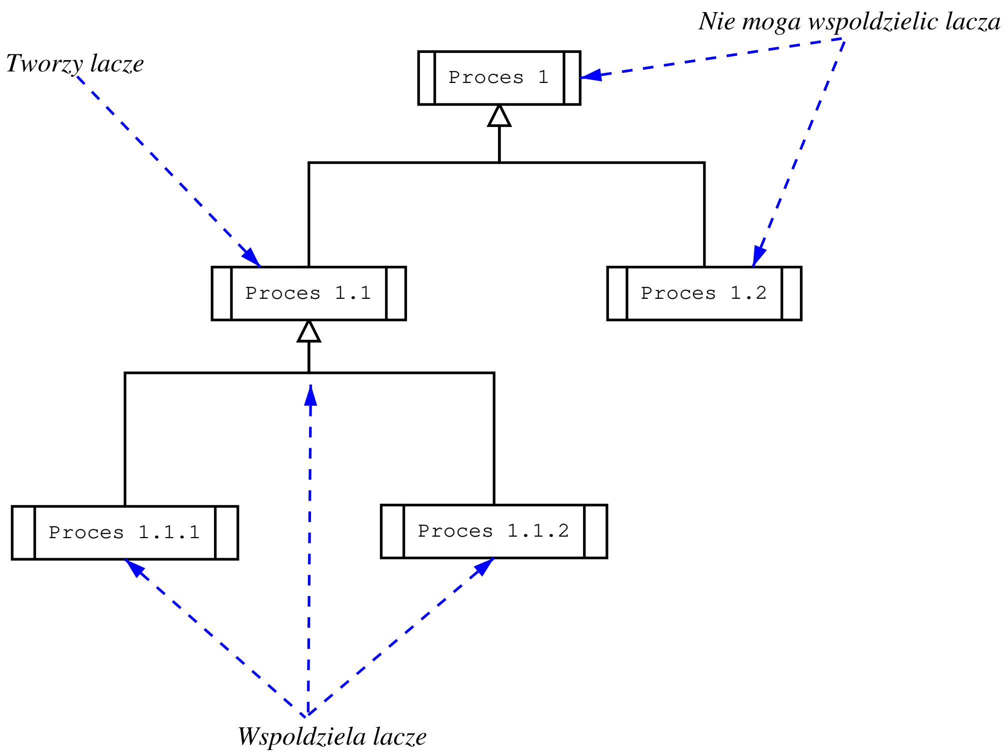
2.1 Powiązania łączy z systemem plików

Zamierzeniem twórców systemu Linux było dostarczenie jak najprostszego interfejsu umożliwiającego wykonywanie operacji na łączach. Dlatego postanowiono zaimplementować mechanizmy dostępu do nich na bazie systemu plików. Zrealizowanie tego projektu okazało się możliwe dzięki dużej elastyczności struktur używanych w tym systemie. Mamy więc dostęp do większości funkcji operujących na "zwykłych" plikach.

Aby to zrealizowac związano z każdym łączem nienazwanym **i-węzeł**. Termin i-węzeł (ang. *inode*) jest skrótem angielskiego terminu *index node*, czyli **węzeł indeksu**. Każde łącze nienazwane (tak jak plik) ma dokładnie jeden odpowiadający mu i-węzeł.

2.2 Struktura i-węzła a łącza

Przyjrzyjmy się dokładniej dwu polom struktury i-węzła.



Rysunek 1: Współdzielenie łączy przez procesy

```
struct inode {
        ...
        off_t i_size;
        ...
        union {
                ...
                struct pipe_inode_info pipe_i;
                ...
        } u;
};
```

Wybierając odpowiedni element **unii u** możemy trzymać informacje zależne od specyfiki obiektu reprezentowanego przez i-węzeł. Dla łączy została stworzona struktura `pipe_inode_info`. W sytuacji, gdy dany i-węzeł reprezentuje łącze, sposród wielu różnych pól unii `u` wybieramy `pipe_i`. W nim to właśnie trzymamy większość danych dotyczących łącza. Z powodu dużej wagi informacji tam zapisanych pole to zasługuje na dokładniejsze omówienie.

2.3 Struktura pipe_inode_info

Przyjrzyjmy się strukturze pipe_inode_info zdefiniowanej w pliku include/linux/pipe_fs_i.h.

```
struct pipe_inode_info {
        struct wait_queue * wait;
        char * base;
        unsigned int start;
        unsigned int lock;
        unsigned int rd_openers;
        unsigned int wr_openers;
        unsigned int readers;
        unsigned int writers;
};
```

Opis poszczególnych pól:

wait Kolejka procesów, które czekają na dostęp do łącza (chcą do niego pisać lub z niego czytać), ale jest to chwilowo niemożliwe (łącze jest puste, a proces chce czytać, łącze jest pełne, a proces chce pisać lub w danej chwili ktoś inny wykonuje jedną z powyższych operacji). W praktyce, przy niewielkich rozmiarach łączy często się zdarza, że dla jakiegoś procesu brakuje miejsca do zapisu lub nie ma danych do odczytania. Umożliwienie czekania na przyjscie danych lub zwolnienie miejsca jest jedną z najważniejszych cech większości mechanizmów komunikacji miedzyprocesowej. Implementacja łączy pozwala też przeprowadzić operacje czytania bądź pisania w trybie nieblokującym (co jest dokładniej opisane w dalszej części).

base Wskaźnik do bufora (położonego na specjalnie dla łącza zarezerwowanej stronie pamięci). To tam pamiętana jest jego zawartość.

start Przesunięcie w buforze (base) odpowiadające początkowi danych. Konieczność posiadania takiej zmiennej wynika z faktu organizacji obszaru na dane jako bufora cyklicznego. Takie rozwiązanie dość dobrze nadaje się do symulacji kolejki FIFO. Mając adres początku bufora (base), przesunięcie (start) i ilość zapisanych danych (patrz niżej) możemy łatwo ustalić położenie obszarów z danymi (może być to jeden lub dwa ciągłe fragmenty pamięci (wirtualnej) - jest to w końcu bufor cykliczny).

lock Służy do blokowania dostępu do łącza. Dzięki niej procesy wiedzą, czy w danej chwili struktur łącza nie modyfikuje ktoś inny.

rd_openers Liczba czytelnikow otwierających łącze. Nie wykorzystywana przy łączach nienazwanych (używana podczas otwierania łącz nazwanych). Dokładny opis pomijamy.

wr_openers J.w. tylko dotyczy liczby pisarzy.

readers Liczba czytelników korzystajacych z łącza (procesów mających łącze otwarte do czytania). Poprzednie stwierdzenie nie jest do końca ścisłe. W przypadku łączy nienazwanych wartość tej zmiennej często nie zgadza się z rzeczywistą liczbą procesów czytających z łącza. Ważne jest natomiast to, że wartość tej zmiennej jest większa od zera jeśli istnieje proces, który ma otwarte łącze do czytania oraz równa zero jeśli takiego procesu nie ma.

writers Liczba pisarzy używających łącza. Uwagi jak wyżej.

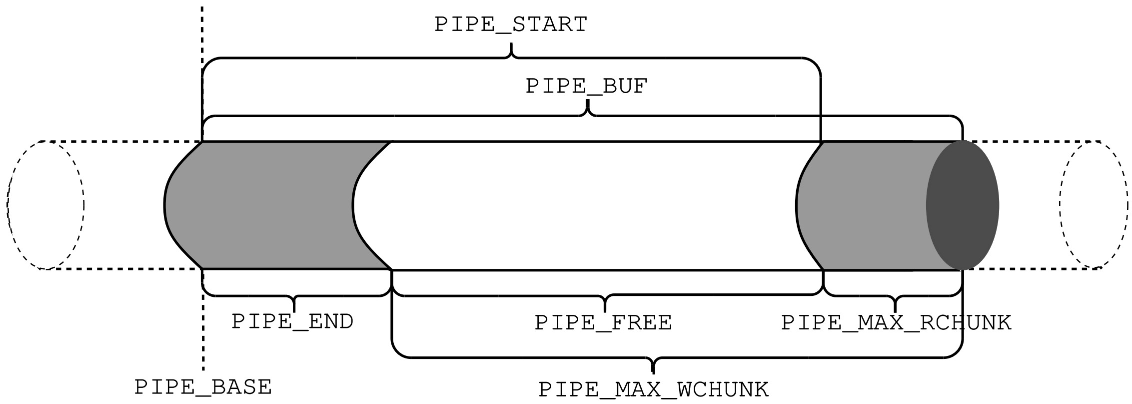
2.4 Rozmiar danych i bufora

Pozostaje jeszcze kwestia pamiętania rozmiaru danych zapisanych w łączu. W starszych wersjach jądra informacja taka była przechowywana w polu `unsigned int len` struktury `pipe_inode_info`. Teraz wartość ta pamiętana jest "poziom wyżej" — w i-węźle łącza (a dokładniej w polu `i_size`).

Wielkość bufora jest pewną ustaloną stałą. Nosi ona nazwę `PIPE_BUF`. W omawianej wersji jądra jej wartość to 4096.

2.5 Pomocnicze makra

W pliku `include/linux/pipe_fs_i.h` znajdziemy też kolekcję makrodefinicji które zostały użyte w kodzie funkcji operujących na łączach. Zrozumienie ich znaczenia pomaga w analizie kodu zródłowego. W wiekszości służą one do ułatwiania obliczania położenia danych w buforze łącza.



Rysunek 2: Bufor łącza

Przyjrzyjmy się dokładniej dwu z nich (o co chodzi w pozostałych łatwo jest się domyślić spoglądając na dołączony rysunek (2)).

```
#define PIPE_MAX_RCHUNK(inode) (PIPE_BUF - PIPE_START(inode))
#define PIPE_MAX_WCHUNK(inode) (PIPE_BUF - PIPE_END(inode))
```

`PIPE_MAX_RCHUNK` Odległość pozycji odpowiadającej początkowi obszaru z danymi od końca bufora. Jeśli łącze byłoby do końca wypełnione danymi to byłaby to maksymalna liczba danych, którą możemy pobrać z bufora przy pomocy jednego wywołania funkcji `copy_to_user()` kopiującej bajty do obszaru procesu (chcąc pobrać więcej danych musimy zacząć ponownie czytać od początku bufora — bufor cykliczny). Makro przydaje się gdy chcemy minimalizować liczbę kopiowań danych[1].

[1] Patrz opis funkcji `pipe_read()`

`PIPE_MAX_WCHUNK` J. w. tylko dotyczy odległości końca obszaru z danymi od końca bufora. Jeśli bufor z danymi byłby do końca pusty to jest to maksymalny rozmiar bloku z danymi, jaki można zapisać w łączu, aby nie wyjść poza bufor (dalej zapisujemy od początku)[2]. Także wprowadzone po to, aby zmniejszyć liczbę kopiowań.

3 Funkcje operujące na łączach

Skoro wiemy już z jakich struktur danych korzystają łącza możemy przejść do algorytmów operujących na nich.

3.1 Tworzenie łącza nienazwanego

Aby móc używać funkcji operujących na plikach także do łączy muszą istnieć odpowiednie wpisy w tablicy deskryptorów. Dlatego w momencie tworzenia, oprócz inicjalizacji i-węzła łączu zostają przydzielone dwa wpisy w tablicy deskryptorów plików procesu pokazujące struktury `file`. Odwołując się do jednej z nich możemy przeprowadzać operacje czytania, a do drugiej — pisania. Obie z tych struktur pokazują na jeden rekord `dentry`, który jest z kolei powiązany z omówionym wcześniej i-węzłem łącza.

Pomimo implementacji na bazie systemu plików organizacja łączy wymusza stworzenie oddzielnych operacji czytania i pisania[3]. `Linux` umożliwia zdefiniowanie dla każdego "pliku" własnych funkcji do operowania jego zawartością. Jest to możliwe dzięki istnieniu bardzo elastycznej struktury `file_operations` w której znajdują się wskaźniki do funkcji odpowiedzialnych za poszczególne operacje plikowe. Wskaźnik na `file_operations` znajduje się w strukturze `file`. Rekordy `file` tworzą tablicę deskryptorów plików danego procesu. Takie rozwiązanie umożliwiło zaimplementowanie łączy tak, że interfejs ich obsługi wygląda jak dla zwyczajnego pliku.

Powiązania między poszczególnymi strukturami łącza pokazano na rysunku 3.

Do tworzenia łącza służy systemowa funkcja `sys_pipe()`. Jej nagłówek wyglada nastepująco:

```
int sys_pipe(unsigned long * fildes)
```

Przekazywany paramert możemy interpretować jako tablicę dwóch deskryptorów: *do czytania* i *do pisania*. Deskryptorów tych używamy potem wykonując operacje czytania lub pisania na łączu. Traktujemy je wtedy jak deskryptory zwykłych plików.

Jak widać (algorytm 1) działanie `sys_pipe()` w zasadzie ogranicza się do wywołania funkcji `do_pipe()`. To tam w rzeczywistosci dokonuje się stworzenie struktur łącza. Jej nagłówek wygląda następująco:

```
int do_pipe(int *fd)
```

Znaczenie argumentu `fd` jest takie samo jak `fildes` w funkcji `sys_pipe()`. Czynności wykonywane przez `do_pipe()` opisuje algorytm 2.

[2]Zobacz też opis funkcji `pipe_write()`
[3]Istnieją jeszcze inne funkcje bedące odpowiednikami operacji plikowych

Algorytm 1 sys_pipe

Dane wejściowe:

`fildes[0]` - deskryptor do czytania
`fildes[1]` - deskryptor do pisania

Wynik:

`0` - jeśli wszystko się udało
`numer błędu` - wpr.

```
Zablokuj jądro;
Wywołaj funkcję do_pipe(), która tworzy struktury łącza;
Odblokuj jądro;
jeśli nie wystapil błąd to
   Kopiuj numery deskryptorów zwrócone przez do_pipe() do fildes;
   Zwróć 0;
w przeciwnym przypadku
   Zwróć błąd;
koniec jeśli
```

Pozostał jeszcze do omówienia mechanizm inicjalizacji i-węzła przydzielonego łączu. Za proces ten jest odpowiedzialna funkcja:

```
struct inode * get_pipe_inode(void)
```

którą z kolej opisuje algorytm 3.

W tabeli 1 opisane są możliwe błędy zwracane w wyniku wywołania `sys_pipe()` uwzględniając sytuacje błędne, do których doprowadziły wywoływane przez nią funkcje pomocnicze.

Tablica 1: Błędy zwracane przez `sys_pipe`

Symbol błędu	Opis błędu
`EFAULT`	Nastąpił błąd podczas zapisu do `fildes`
`ENFILE`	W systemie jest za dużo otwrtych plików i nie udało się nam zdobyć wolnej struktury `file`, nie dostalismy i-węzła badź nie powiodła się próba przydzielenia strony w pamięci na bufor łącza
`EMFILE`	Nie udało się uzyskać wolnego deskryptora pliku
`ENOMEM`	Nie ma wystarczająco dużo pamięci

Algorytm 2 do_pipe

Dane wejściowe:

`fd[0]` - deskryptor do czytania
`fd[1]` - deskryptor do pisania

Wynik:

`0` - jeśli wszystko się udało
`numer błędu` - wpr.

1: Pobierz dwie puste struktury `file` (oznaczmy je przez `f1` i `f2`). `F1` będzie odpowiadała deskryptorowi do czytania, natomiast `f2` — deskryptorowi do pisania do łącza;
2: Pobierz i zainicjalizuj pusty i-węzeł wywołując w tym celu funkcję `get_pipe_inode()`;
3: Pobierz dwa wolne deskryptory plików (`i` oraz `j`);
4: Powiąż `f1` i `f2` z i-węzłem łącza za pośrednictwem `dentry`. Struktura `dentry` jest wspólna dla `f1` oraz `f2`;
5: Ustaw dla f1 i f2 pozycję w pliku na 0;
6: Ustaw tryb f1 *do czytania* (== `1` == `FMODE_READ`);
Ustaw flagę *tylko do czytania* (`O_RDONLY`) w `f1`;
Powiąż `f1` z odpowiednimi operacjami (strukturą `file_operations` zawierającą wskaźniki do funkcji operujących na łączach);
7: Ustaw tryb f2 *do pisania* (== `2` == `FMODE_WRITE`);
Ustaw flagę *tylko do pisania* (`O_WRONLY`) w `f2`;
Powiąż `f2` z odpowiednimi operacjami (struktura `file_operations` dla `f2` jest nieco inna niż dla `f1`);
8: Wpisz `f1` i `f2` do tablicy deskryptorów plików procesu pod indeksami `i` oraz `j`;
9: Zapisz `i` oraz `j` w `fd`;

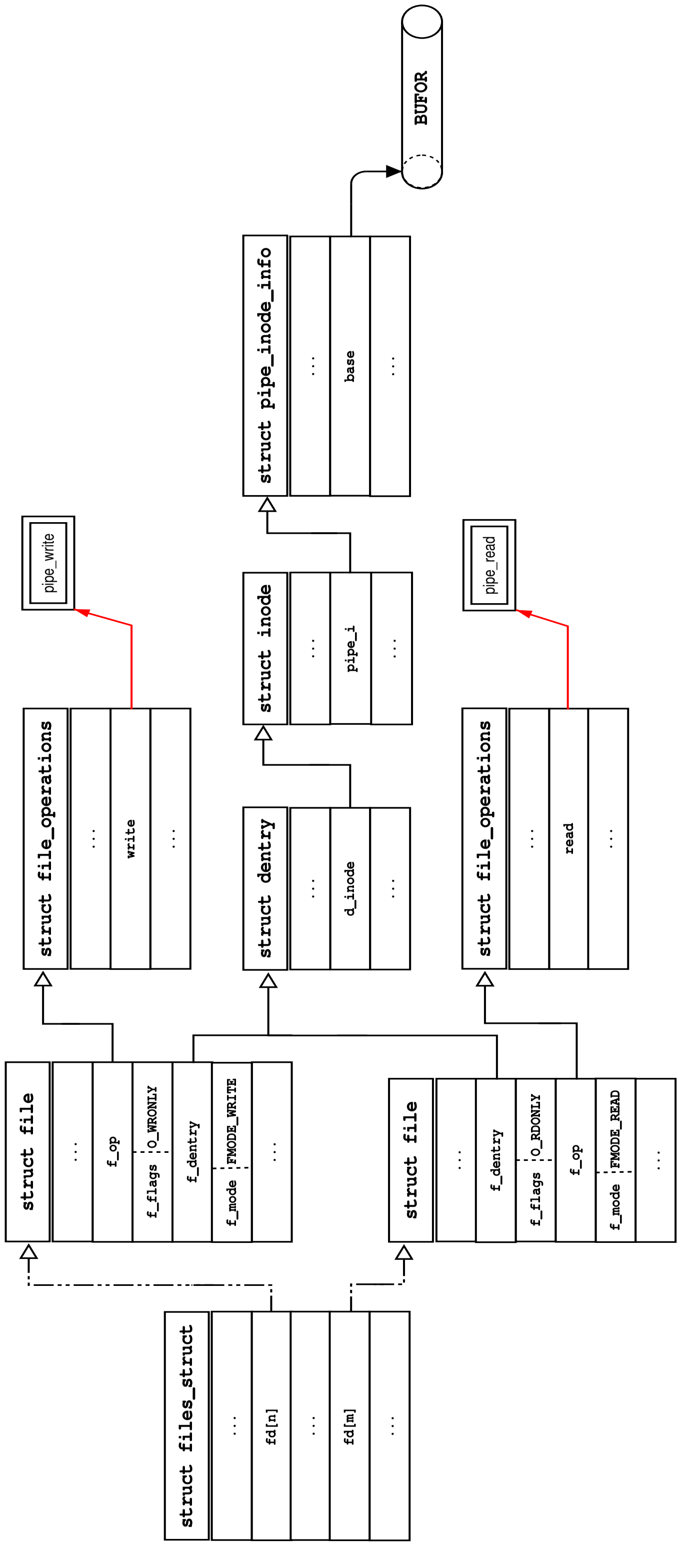
Algorytm 3 get_pipe_inode

Dane wejściowe:

Wynik:

Wskaźnik do zainicjalizowanego i-węzła reprezentującego łącze.

1: Pobierz pusty i-węzeł;
2: Zarezerwuj wolną stronę w pamięci (na bufor łącza). Mając na uwadze fakt, że czyszczenie całej strony przed jej przydzieleniem trwa dosyć długo, używamy funkcji `__get_free_page()`, która omija procedurę czyszczenia. Dla nas będzie wszystko jedno czy na stronie wcześniej znajdowały się jakieś informacje czy nie, bo sami pamiętamy położenie zapisanych przez nas danych;
3: Ustaw w i-węźle adres bufora łącza na przydzieloną stronę, przesunięcie początku i liczbę danych na 0, liczbę czytelników i pisarzy na 1, blokadę na 0 (można uzyskać dostęp do łącza). Dokonujemy też inicjalizacji pól nie dotyczących bezpośrednio łącza. Ustawiamy w polu trybu pliku flagę `S_IFIFO` (nie ma specjalnej flagi oznaczającej łącze nienazwane) oraz prawa dostępu, datę utworzenia, modyfikacji, identyfikator właściciela, grupy .... Dosyć ważne jest to, że zaznaczamy i-węzeł od samego początku jako brudny (ustawiamy `i_state` na `I_DIRTY`). Efekt jest taki, że nie zostanie on nigdy przeniesiony do listy brudnych i-węzłów, bo funkcja która za to odpowiada (`mark_inode_dirty()`) będzie myślała, że skoro ma ustawioną flagę to pewnie już się tam znajduje;



Rysunek 3: Struktury danych łącza

3.2 Operacja czytania z łącza

Funkcja, która realizuje czytanie z łącza to:

```
ssize_t pipe_read(struct file * filp, char * buf, size_t count, loff_t *ppos)
```

Gwarantujemy, że operacja czytania jest *niepodzielna*. Nie dajemy pewności, że przeczytamy tyle danych ile zażadaliśmy (nigdy nie przeczytamy za dużo, a najwyżej za mało). Umożliwiamy też przeprowadzenie operacji w trybie *blokującym* lub *nieblokującym*. W skrócie polega to na tym, że jeśli czytamy nieblokująco i w buforze nic nie ma, to natychmiast wychodzimy. Jeśli weszliśmy w trybie blokującym to czekamy, aż pojawią się dane (lub nastąpi jakieś inne zdarzenie np. wszyscy pisarze zamkną łącze — wtedy nie ma co dalej czekać).

Dokładnie dzialanie funkcji opisuje algorytm 4.

Uwagi do algorytmu 4:

Dane wejściowe Parametr `ppos` nie jest w zasadzie przez nas wykorzystywany. Sprawdzamy tylko jego poprawność. Został on uwzględniony w nagłówku w celu uzyskania zgodności z szablonem funkcji `read()` dla "zwykłych" plików. W ogólnym przypadku parametr `ppos` oznacza pozycję w pliku (pamiętana w strukturze `file`). Nam nie jest to do niczego potrzebne, bo do określenia aktualnego położenia danych wykorzystujemy pola `pipe_inode_info`.

Linia 4 Po co tracić czas na dostęp do łącza skoro i tak niczego nie będziemy czytać — możemy wyjść już teraz.

Linia 14 Nie ma (i nigdy nie będzie, bo tego samego łącza nienazwanego nie można powtórnie otworzyć) żadnych pisarzy. Wychodzimy zwracając 0, a nie kod jakiegoś błędu.

Linia 30 Staramy się zminimalizować liczbę kopiowań w czym pomaga nam opisywane wcześniej makro `PIPE_MAX_RCHUNK`.

Tablica 2: Błędy zwracane przez `pipe_read`

Symbol błędu	Opis błędu
`ESPIPE`	Pozycja w pliku przekazana do funkcji nie zgadza się z tą zapisaną w strukturze `file`
`EAGAIN`	Nie udało się uzyskać natychmiastowego dostępu do bufora łącza lub nie było żadnych danych do odczytania (mimo, że istnieją procesy mające otwarte łącze do pisania) podczas próby odczytu w trybie nieblokującym, badź też nic nie przeczytaliśmy mimo iż są pisarze i w łączu powinny znajdować się dane
`ERESTARTSYS`	Otrzymaliśmy sygnał

Algorytm 4 pipe_read

Dane wejściowe:

`filp` - wskaźnik do struktury `file` odpowiadającej łączu komunikacyjnemu i umieszczonej w tablicy plików
`buf` - wskaźnik do bufora zawierającego dane, które mamy odczytać z łącza
`count` - liczba bajtów do wczytania z łącza
`ppos` - pozycja w pliku

Wynik:

`liczba przeczytanych bajtów` - jeśli wszystko się udało
`numer błędu` - wpr.

```
Pobieramy i-węzeł skojarzony z łączem;
Sprawdzamy poprawność parametru ppos (czy wartość, którą mamy zgadza się z wartością
  odpowiedniego pola w filp);
jeśli nic nie chcemy czytać (count == 0) to
   Wychodzimy zwracając 0;
koniec jeśli
jeśli czytamy w trybie nieblokującym to
   jeśli nie ma natychmiastowego dostępu do łącza to
      Wychodzimy z błędem EAGAIN;
   koniec jeśli
   jeśli łącze jest puste to
      jeśli z łączem związani są jacyś pisarze to
         Wychodzimy z błędem EAGAIN;
      w przeciwnym przypadku
         Wychodzimy zwracając 0;
      koniec jeśli
   koniec jeśli
w przeciwnym przypadku
   dopóki bufor łącza jest pusty lub nie ma do niego dostępu wykonuj
      jeśli łącze jest puste i nie ma pisarzy to
         Wychodzimy zwracając 0;
      koniec jeśli
      jeśli dostaliśmy sygnał to
         Wychodzimy z błędem ERESTARTSYS;
      koniec jeśli
      Zawieszamy się na kolejce związanej z łączem;
   koniec dopóki
koniec jeśli
Blokujemy dostęp do łącza;
dopóki mamy jeszcze coś do odczytania i w buforze łącza jeszcze coś jest wykonuj
   Przenosimy dane z bufora łącza do pamięci użytkownika. Po zakończeniu kopiowania uak-
     tualniamy informacje dotyczące ilości bajtów w łączu, przesunięcia początku danych, liczby
     bajtów którą mamy jeszcze odczytać;
koniec dopóki
Odblokowujemy dostęp do łącza;
Budzimy procesy czekające na dostęp do bufora;
jeśli coś zostało przeczytane to
   Uaktualniamy czas ostatniego dostępu do i-węzła;
   Zwracamy liczbę przeczytanych bajtów;
koniec jeśli
jeśli nic nie przeczytaliśmy, mimo iż są pisarze i w łączu powinny być dane to
   Wychodzimy z błędem EAGAIN;
koniec jeśli
Zwracamy 0;
```

3.3 Operacja pisania do łącza

Funkcja realizująca tę operację to:

```
static ssize_t pipe_write(struct file * filp, const char * buf,
  size_t count, loff_t *ppos)
```

W przypadku tej funkcji nie zawsze możemy zagwarantować niepodzielność zapisu. Jeśli liczba danych do zapisania przekracza rozmiar bufora łącza, to zapis może dokonać się w kilku krokach, tak więc dane mogą się potem przemieszać przy jednoczesnym odczycie przez kilka procesów. Jeśli natomiast wielkość informacji do wrzucenia do łącza nie przekracza jego rozmiaru, to gwarantujemy niepodzielność.

Tak jak poprzednio mamy dwa możliwe tryby zapisu: *blokujący* i *nieblokujący*.

Uwagi do algorytmu 5:

Linia 4 i 17 Sygnał `SIGPIPE` oznacza tu próbę zapisu do łącza przy braku szansy na to, że kiedykolwiek ktoś będzie z niego czytał. Jesli nie sprawdzalibyśmy czy są jeszcze czytelnicy, a w łączu nie byłoby wystarczająco dużo miejsca na zapis to moglibyśmy zostać zablokowani "na zawsze" — czekalibyśmy ma przyjście danych, których nikt by nie mógł zapisać. Zachodziłaby taka możliwość tylko podczas pisania w trybie blokującym.

Linia 10 Będziemy czekać na jakiekolwiek wolne miejsce w buforze.

Linia 12 Ten semafor został opuszczony tuż przed wywołaniem `pipe_write()`[4]. Podniesienie go w tym miejscu umożliwia innym pisarzom wykonywanie czynności wstępnych, gdy my będziemy np. czekać na pozwolenie pisania;

Linia 13 Odtąd przed wyjściem z funkcji (spowodowanym błędem lub po wykonaniu zapisu) będziemy musieli po sobie posprzątać, czyli przywrócić poprzedni stan semaforom (podnoszenie semafora jest zwykle natychmiastowe, ale opuszczanie może trochę potrwać). To tłumaczy dlaczego zajęliśmy się operacjami na semaforach dopiero po sprawdzeniu niektórych sytuacji błędnych;

Linia 18, 21, 24 Przed wyjściem sprawdzamy, czy coś udało się zapisać i jeśli tak to zamiast błędu zwracamy liczbę zapisanych bajtów (tak naprawdę skaczemy do linii 37).

Linia 30 Tu przydaje się makro `PIPE_MAX_WCHUNK`.

Tablica 3: Błędy zwracane przez `pipe_write`

Symbol błędu	Opis błędu
`EPIPE`	Nie istnieje proces, który ma otwarte łącze do czytania
`EAGAIN`	Nie udało się uzyskać natychmiastowego dostępu do bufora łącza lub nie było wystarczająco dużo wolnego miejsca podczas próby zapisu w trybie nieblokującym
`ERESTARTSYS`	Nie udało się opuścić semafora do pisania na i-węźle lub dostaliśmy sygnał

[4]patrz `fs/read_write.c` funkcja `sys_write()`

Algorytm 5 pipe_write

Dane wejściowe:

`filp` - znaczenie parametru jak w funkcji `pipe_read()`
`buf` - bufor z danymi do wsadzenia do łącza
`count` - liczba bajtów do przesłania do łącza
`ppos` - znaczenie parametru jak w funkcji `pipe_read()`

Wynik:

`liczba zapisanych bajtów` - jeśli wszystko się udało
`numer błędu` - wpr.

Pobieramy i-węzeł skojarzony z łączem;
Sprawdzamy poprawność parametru `ppos` (czy wartość, którą mamy zgadza się z wartością odpowiedniego pola w `filp`);
jeśli nie ma czytelników **to**
  Wysyłamy sygnał `SIGPIPE`;
  Wychodzimy z błędem `EPIPE`;
koniec jeśli
jeśli wielkość danych, którą mamy zapisać nie przekracza rozmiaru bufora łącza **to**
  Żądamy aby zapis dokonywał się atomowo. Uzyskujemy to przez przypisanie zmiennej oznaczającej wielkość wolnego miejsca na które czekamy wartości `count`;
w przeciwnym przypadku
  Nie można dokonać zapisu atomowo, więc będziemy zapisywać do łącza nawet po jednym bajcie;
koniec jeśli
Podnosimy semafor i-węzła;
Opuszczamy semafor do pisania na i-węźle łącza — wpuszczamy tylko jednego pisarza;
dopóki mamy jeszcze coś do zapisania **wykonuj**
  dopóki w łączu nie ma wystarczająco dużo wolnego miejsca lub dostęp jest zablokowany **wykonuj**
    jeśli nie ma czytelników **to**
      Wysyłamy sygnał `SIGPIPE`;
      Wychodzimy z błędem `EPIPE`;
    koniec jeśli
    jeśli dostaliśmy sygnał **to**
      Wychodzimy z błędem `ERESTARTSYS`;
    koniec jeśli
    jeśli piszemy w trybie nieblokującym **to**
      Wychodzimy z błędem `EAGAIN`;
    koniec jeśli
    Zasypiamy na kolejce łącza;
  koniec dopóki
  Blokujemy dostęp do łącza;
  dopóki mamy coś do zapisania i jest wolne miejsce **wykonuj**
    Dokonujemy zapisu starając się zapisać za jednym razem najwięcej ile można. Aktualizujemy przy tym pozycję początku i ilość danych w buforze cyklicznym, wielkość zapisanych danych;
  koniec dopóki
  Odblokowujemy dostęp do łącza;
  Budzimy procesy czekające na dostęp;
koniec dopóki
Ustawiamy czas utworzenia oraz modyfikacji i-węzła łącza na czas aktualny;
Zaznaczamy i-węzeł jako brudny;
Wychodzimy zwracając liczbę zapisanych bajtów lub błąd jeśli niczego nie zapisaliśmy;

4 Informacje dodatkowe

Znajduje się tu kilka krótkich opisów zagadnień pominiętych w poprzednich działach.

Inne operacje na łączach Oprócz `read()` i `write()` na łaczach nienazwanych można przeprowadzać także inne operacje charakterystyczne dla systemu plików. Jako przykład można podać `pipe_lseek()` będącą opdowiednikiem funkcji `lseek()`. Jak możemy się spodziewać jej działanie polega na zwróceniu błędu przeszukania (`ESPIPE`) — z łącz można korzystać tylko w trybie sekwencyjnym.

Zwalnianie struktur łącza Do tej pory nie powiedzieliśmy w którym momencie następuje zmniejszenie liczby czytelników/pisarzy związanych z łączem. Otóż w każdej strukturze `file` znajduje się pole `f_count`, gdzie pamiętamy liczbę odwołań do tego rekordu. Liczba ta jest zwiększana np. po wywołaniu funkcji `dup()`. Podczas zamykania deskryptora łącza (po prostu wywołując `close()`) zminiejszamy licznik odwołań. Kiedy `f_count` spadnie do 0 system uruchamia `release()`. Wskaźnik na funkcję `release()` znajduje sie w strukturze z operacjami. Dla łączy zdefiniowano dwie wersje tej funkcji — zależną od tego czy zamykamy deskryptor do pisania czy do czytania. `Release()` zmniejsza liczbę pisarzy bądź czytelników związanych z łączem. Jeśli obie te liczby wskarzą zero następuje zwolnienie struktur łącza.

Pliki Nazwy plików źródłowych do których warto zajrzeć:

- `include/linux/fs.h`
- `include/linux/pipe_fs_i.h`
- `fs/pipe.c`
- `fs/open.c`
- `fs/read_write.c`
- `fs/file_table.c`

5 Zadania

1. Jak wiadomo bufor łącza zaimplementowano jako bufor cykliczny. W takim przypadku dość często wykonujemy operację modulo `PIPE_BUF`. *Przypomnienie :* `BUF_SIZE` to rozmiar bufora. Ze względu na efektywność chcielibyśmy móc obliczyć szybko wynik takiej operacji. Zastanów się jak efektywnie $\forall y \geq 0$ obliczyć x taki, że

$$x \in [0, \mathtt{PIPE\_BUF} - 1] \quad \wedge \quad x \equiv y \pmod{\mathtt{PIPE\_BUF}}$$

   Wskazówka: zastanów się jakie znaczenie ma to, że `PIPE_BUF` jest potęgą dwójki.

2. Zastanów się ile razy może "obrócić" się pętla z linii 29 - 31 algorytmu 4.

6 Rozwiązania zadań

1. Rozpatrzmy liczbę $a = (\texttt{PIPE\_BUF} - 1)$. W sytuacji, gdy PIPE_BUF jest potęgą dwójki a w zapisie binarnym ma postać $a = 1\ldots$ (same jedynki). Obliczając więc a & $(\texttt{PIPE\_BUF} - 1)$ otrzymamy to o co chodzi. Logiczne operacje bitowe są bardzo efektywne — dzięki zastosowaniu tego chwytu zaoszczędzamy trochę czasu procesora. Pomysł ten wykorzystano np. w makrze PIPE_END obliczającym przesunięcie odpowiadające końcowi obszaru z danymi w buforze.

2. Algorytm stara się zminimalizować liczbę kopiowań danych (w tym celu zdefiniowano przecież makro PIPE_MAX_RCHUNK). Dlatego też jeśli dane są umieszczone w buforze łącza w ciagłym obszarze pamieci wirtualnej to pętla wykona się jeden raz. Jeśli natomiast dane są rozdzielone na dwa rozłączne obszary to bedą dwa obroty pętli.

Bibliografia

[Bach 95] Bach M. J.: *Budowa systemu operacyjnego Unix.* Warszawa, WNT 1995.

[Rochkind 93] Rochkind M. J.: *Programowanie w systemie Unix dla zaawansowanych.* Warszawa, WNT 1993.

[SilbGalv 2000] Silberschatz A., Galvin P. B.: *Podstawy systemów operacyjnych.* Warszawa, WNT 2000.