Systemy operacyjne

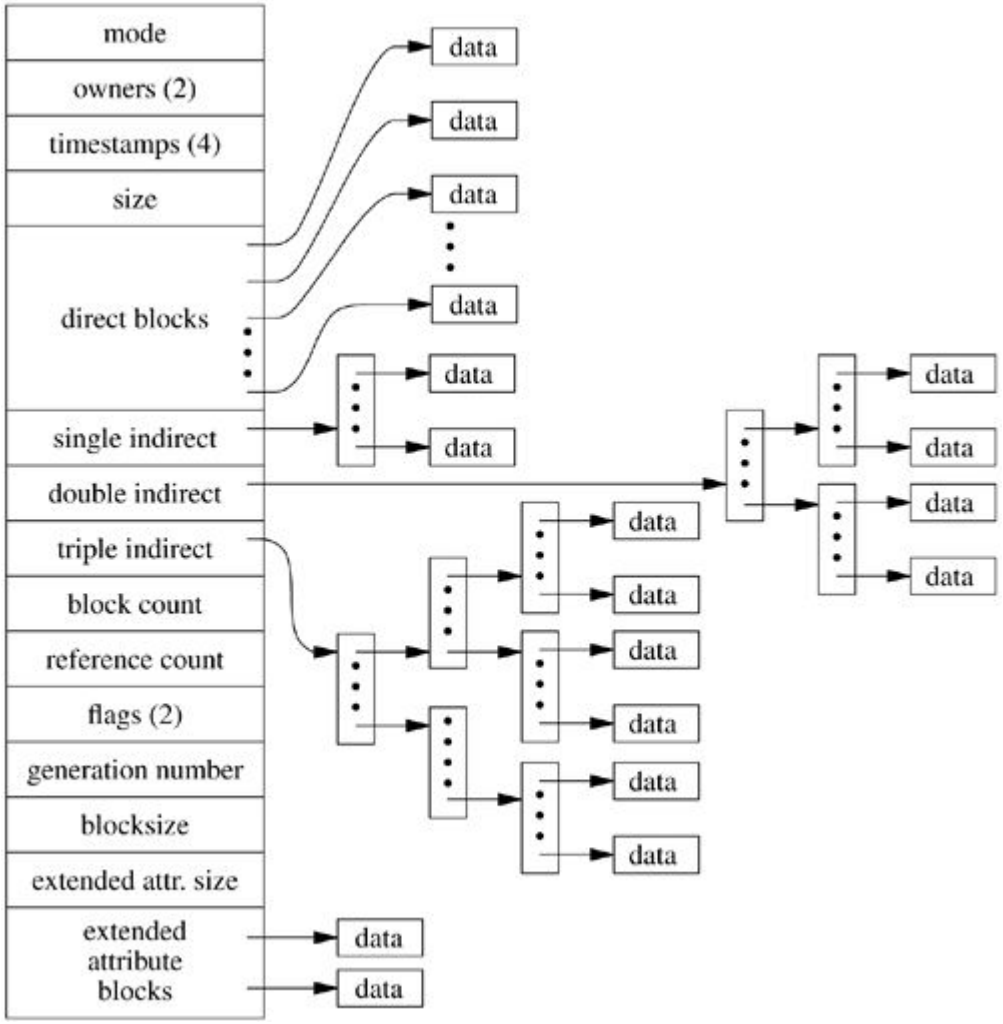
(slajdy uzupełniające)

Wykład 5: Pliki (c.d.)

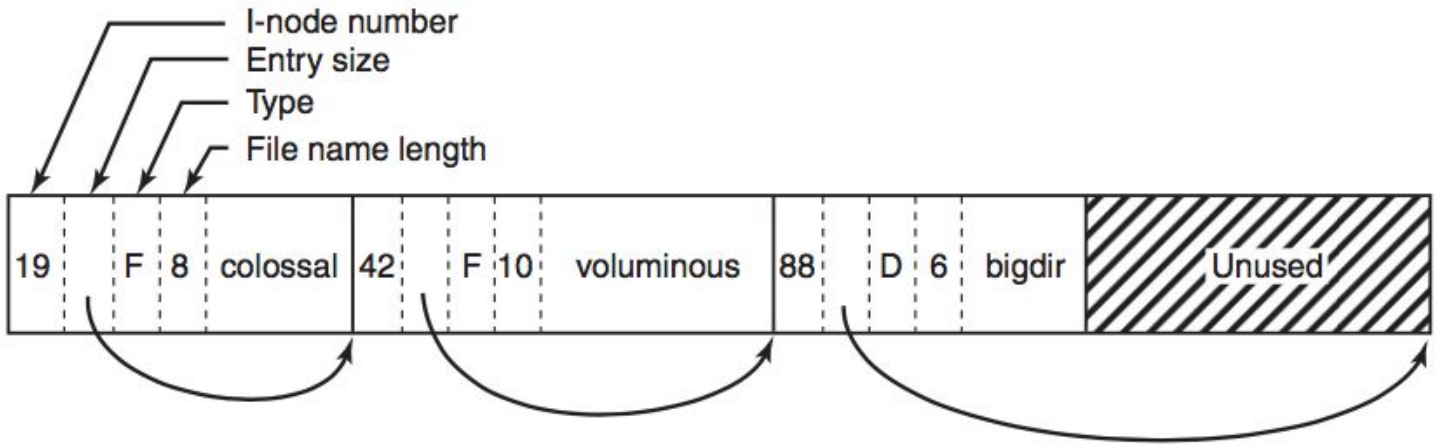
i-węzeł (ang. *i-node*)

Opis zasobu dyskowego.
Oprócz atrybutów zawiera wskaźniki na bloki danych i **bloki pośrednie** (ang. *indirect blocks*).

Przydział bloków w strukturze drzewiastej, która rośnie wraz z rozmiarem pliku.



Reprezentacja katalogów



Reprezentacja listowa → liniowe wyszukiwanie. Dodawanie, usuwanie zmiana nazwy plików potencjalnie wymaga przejrzenia całego katalogu.

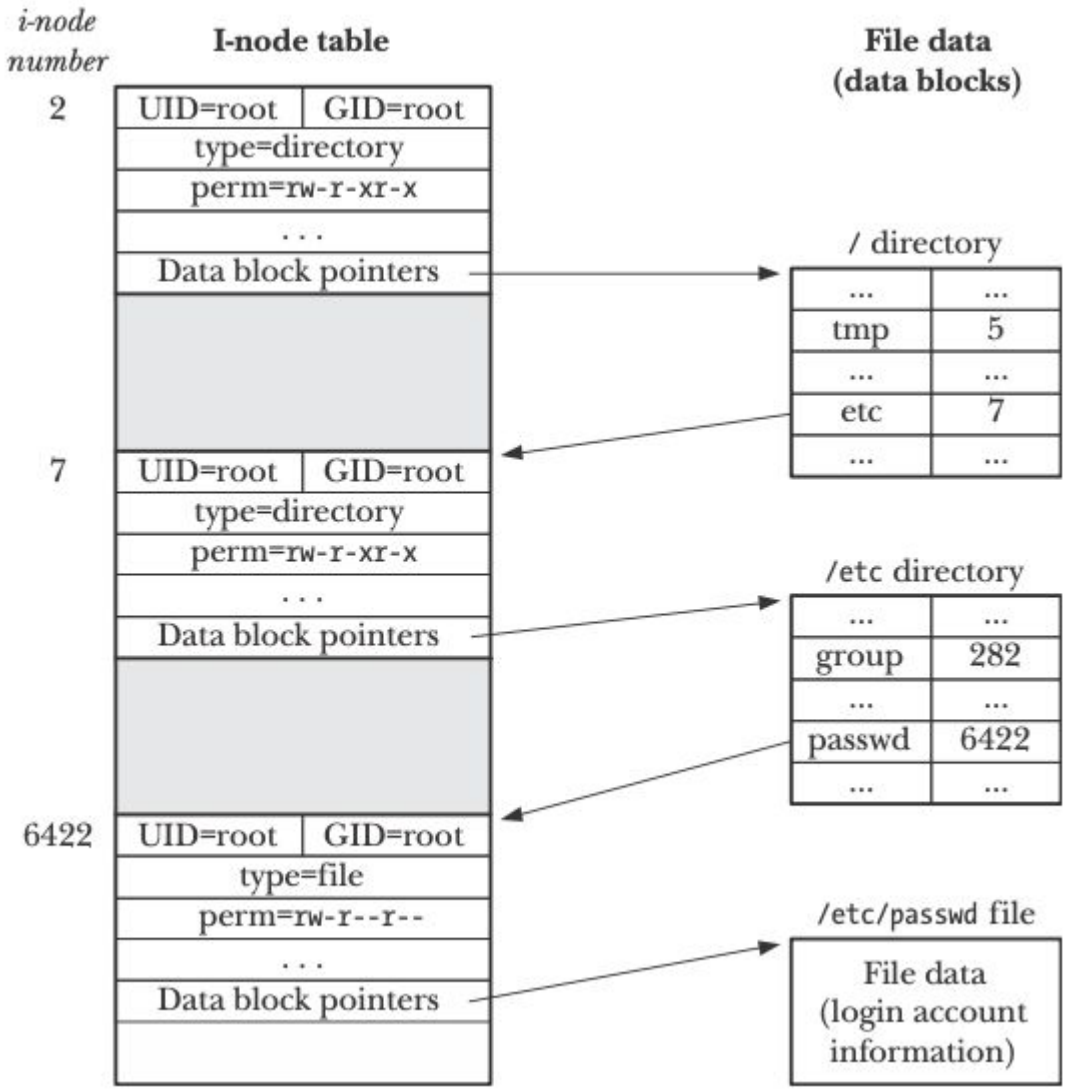
W wyniku operacji na katalogu w reprezentacji powstają nieużytki → rozmiar wpisu może być dużo większy niż nazwa pliku przechowywanego przez wpis. Co jakiś czas potrzebne **kompaktowanie**, które zmniejsza rozmiar katalogu i potencjalnie zwalnia nieużywane bloki na końcu.

Przechodzenie ścieżki (źródło: LPI 18-1)

System plików dysponuje tablicą wszystkich i-węzłów.

Przechodzenie ścieżki zaczyna się od katalogu głównego, którego i-węzeł ma numer 2.

Jądro odczytuje dane katalogu i wyszukuje pary (nazwa, #i-węzła).



Dowiązania symboliczne

```
int symlink(const char *target, const char *linkpath);
```

```
ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

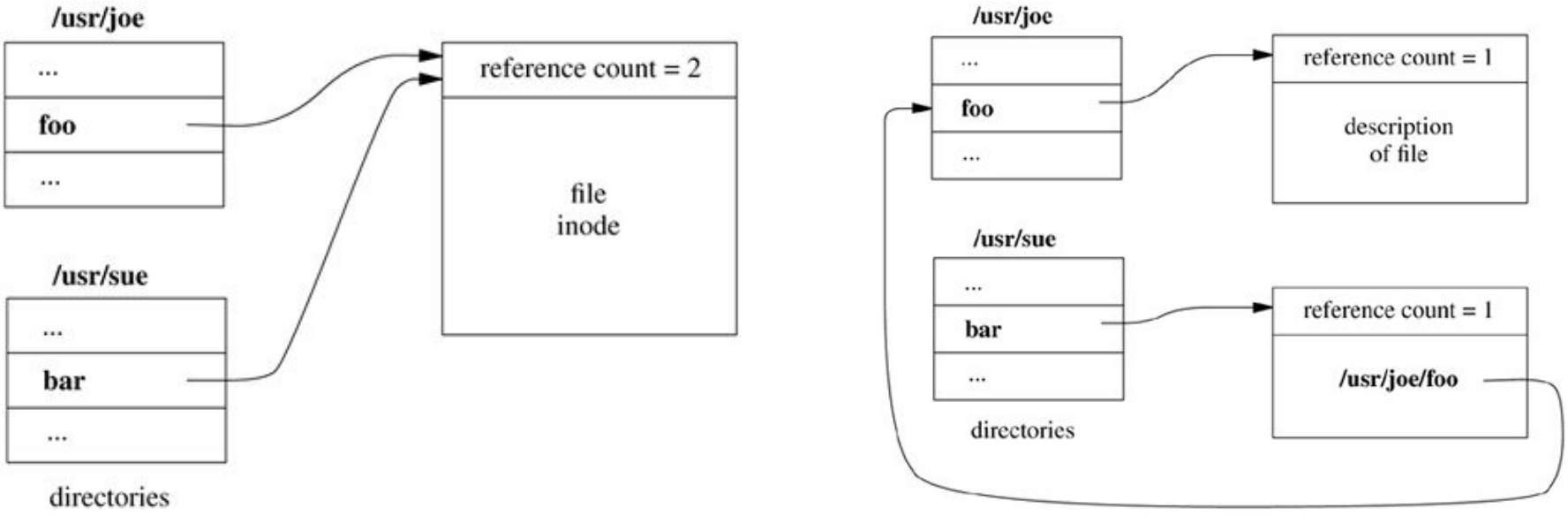
Dowiązania symboliczne (ang. *symbolic links*) specjalny typ pliku, który w zawartości przechowuje ścieżkę do innego pliku. System nie sprawdza poprawności tej ścieżki → może powstać pętla.

Działa jak słaba referencja → plik docelowy może przestać istnieć, system dopuszcza **wiszące dowiązania** (ang. *dangling symlinks*).

Dereferencja dowiązania jest przezroczysta. Nie wykonujemy operacji na pliku dowiązania tylko na tym na co wskazuje. Zawsze?

Problem na poziomie API! Jak pobrać właściwość dowiązania zamiast pliku docelowego? Funkcje z prefiksem `l`, np. `lstat`.

Dowiązanie symboliczne vs. twarde



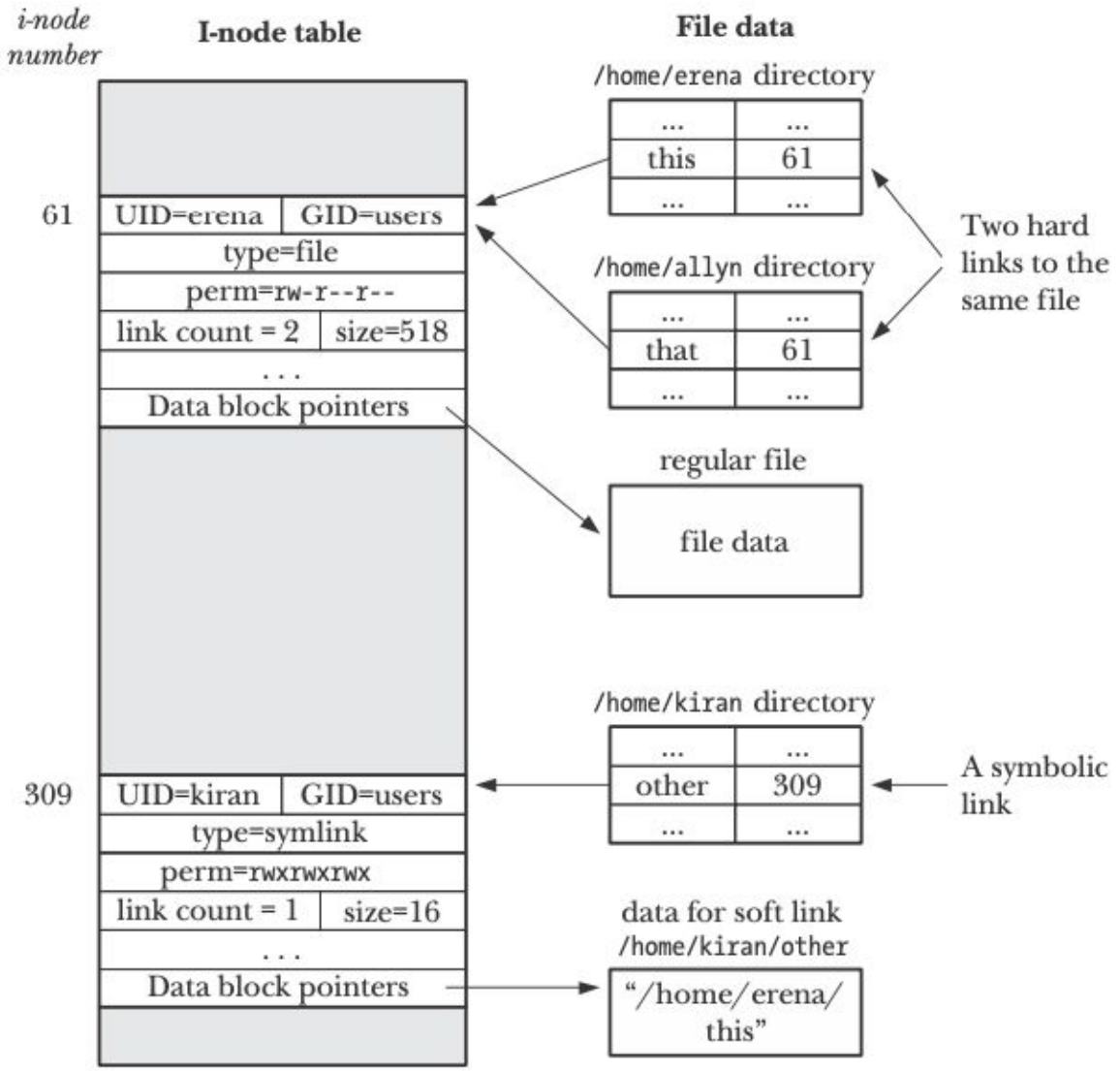
Dowiązania twarde to wskaźniki na i-węzły (licznik referencji!) plików → różne nazwy tego samego pliku w obrębie jednego systemu plików.

Dowiązania symboliczne kodują ścieżkę do której należy przekierować algorytm rozwiązywania nazw.

Różnice między dowiązaniami (źródło: LPI 18-2)

W katalogach: **`/home/erena`** i **`/home/allyn`** mamy dwie nazwy z tym samym #i-węzła → dowiązanie twarde.

Dowiązanie symboliczne **`/home/kiran/other`** restartuje przeglądanie ścieżki (zaczyna się od “**`/`**”).



Rury

Jednokierunkowe (klasyczny Unix i Linux) lub dwukierunkowe (FreeBSD, MacOS) **strumieniowe** przesyłanie danych z buforowaniem w jądrze.

Rury zachowują się przy odczycie jak zwykłe pliki
→ *short count* tylko, jeśli nie ma więcej danych.
Przy zapisie jest ciekawiej, do długości zapisu **`PIPE_BUF`**
`write` dopisuje do bufora atomowo, tj. w jednym kroku.

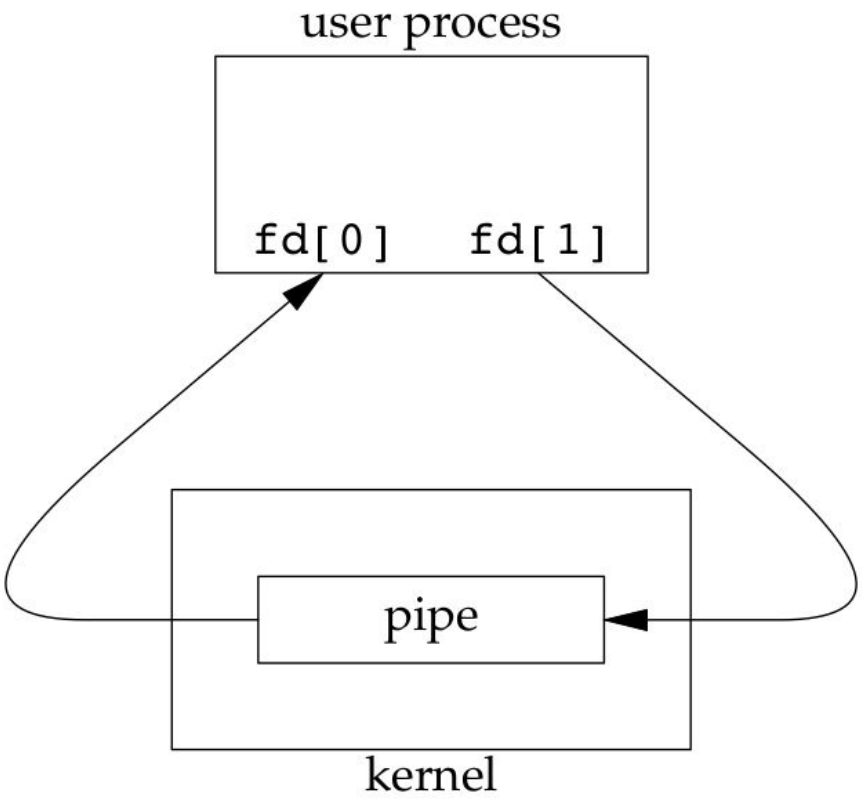
Nazwane rury, tj. takie które posiadają nazwę w systemie plików, nazywamy FIFO (mkfifo).

Potoki występują również w WindowsNT.

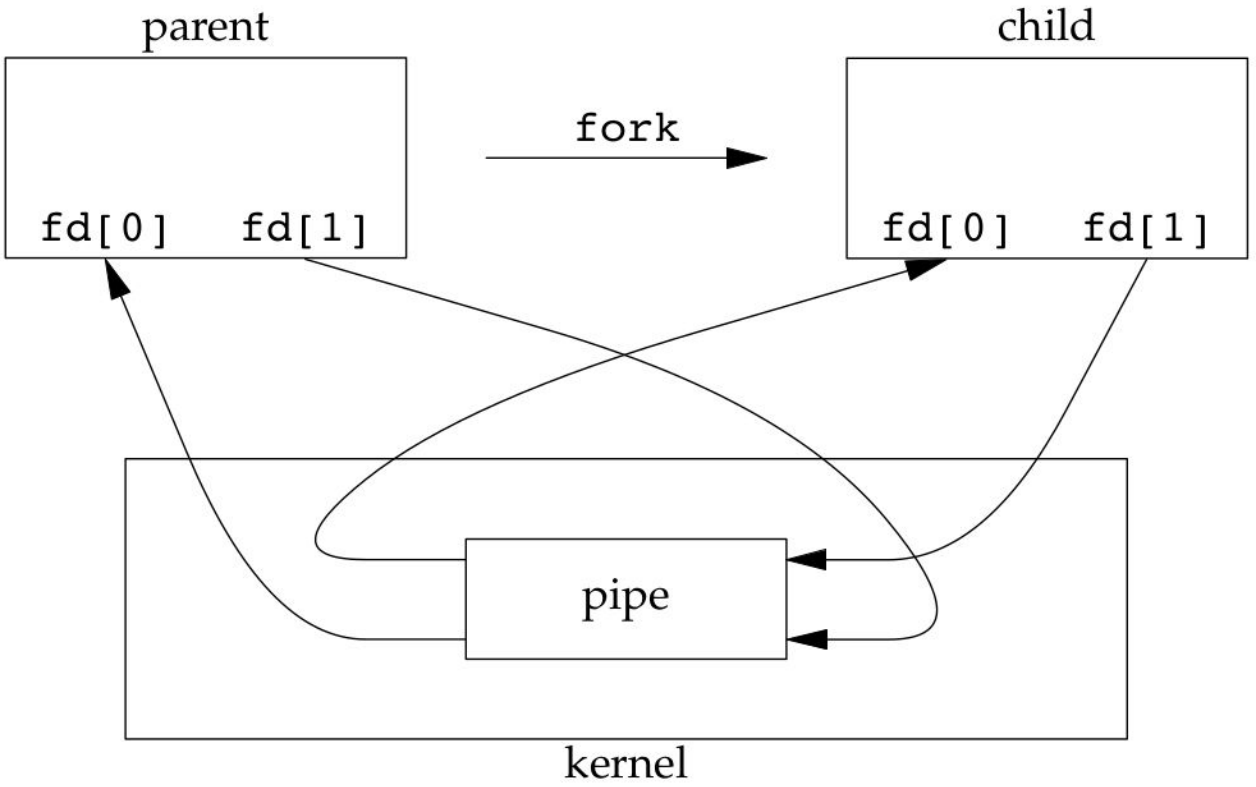
Potoki: przykład

```
int main(void) {
  int fd[2];
  Pipe(fd);
  if (Fork()) { /* parent */
    Close(fd[0]);
    Write(fd[1], "hello world\n", 12);
  } else {     /* child */
    char line[MAXLINE];
    Close(fd[1]);
    int n = read(fd[0], line, MAXLINE);
    write(STDOUT_FILENO, line, n);
  }
  return EXIT_SUCCESS;
}
```

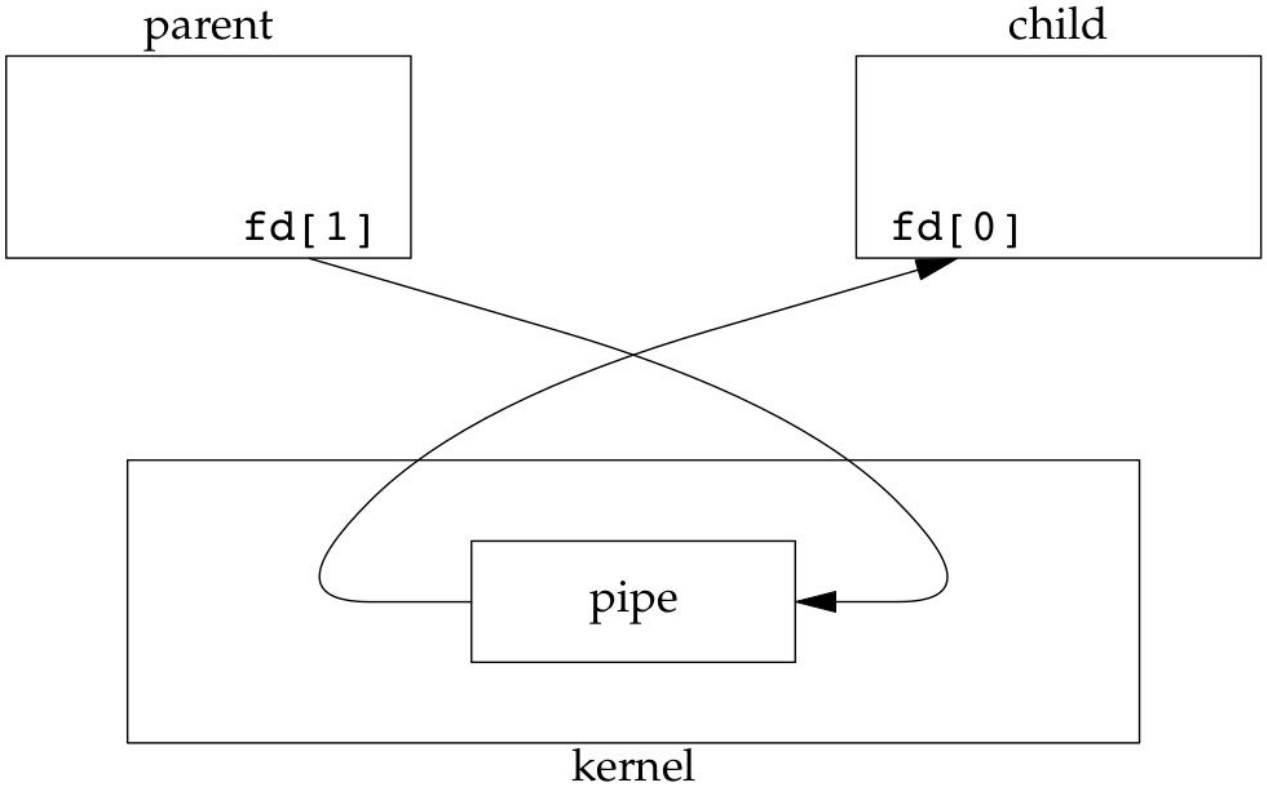
Etap 1: tworzenie rury



Etap 2: wykonanie `fork()`



Etap 3: Zamknięcie niepotrzebnych końców



Potoki aka rurociągi

Przy pomocy powłoki użytkownik może potok – zestaw procesów komunikujących się przy pomocy rur.

```
cat /dev/random | tr -cd 'a-zA-Z0-9' | head -c 16
```

1. `cat` produkuje na `stdout` nieskończony ciąg bajtów
2. `tr` czyta z `stdin` nieskończony ciąg znaków ASCII, usuwa z niego znaki nienależące do danego zbioru i drukuje na `stdout`
3. `head` konsumuje do 16 znaków z `stdin` i drukuje je na `stdout` po czym kończy swe działanie

Jak to się dzieje, że `cat` i `tr` nie działają w nieskończoność?

Koordynacja pracy procesów przy pomocy rur

Każda rura ma skończony cykliczny bufor w jądrze rozmiaru kilku stron pamięci. Jeśli bufor rury jest pełny / pusty to odpowiednio zapis do / odczyt z rury zablokują proces.

Jeśli zostanie zamknięty koniec do:

- zapisu, to konsument ostatecznie opróżni bufor rury po czym dostanie `EOF` (`read` zwróci 0)
- odczytu, to producent przy zapisie dostanie `SIGPIPE`, którego domyślną akcją jest zakończenie procesu

Po tym jak `head` skończy działanie `tr` dostanie `SIGPIPE` i zginie. To samo stanie się z `cat`. Widać, że kod wyjścia potoku powinien być kodem wyjścia z `head`, czyli ostatniego procesu w potoku!

Gniazda domeny uniksowej

Dwukierunkowa metoda komunikacji lokalnej. Przesyłanie strumieniowe (**`SOCK_STREAM`**), datagramowe (**`SOCK_DGRAM`**) lub sekwencyjne pakietowe (**`SOCK_SEQPACKET`**).

Nienazwane gniazda tworzymy `socketpair`.

Dla gniazd typu **`DGRAM`** i **`SEQPACKET`** jądro zachowuje granice między paczkami danych (zwanych datagramami).

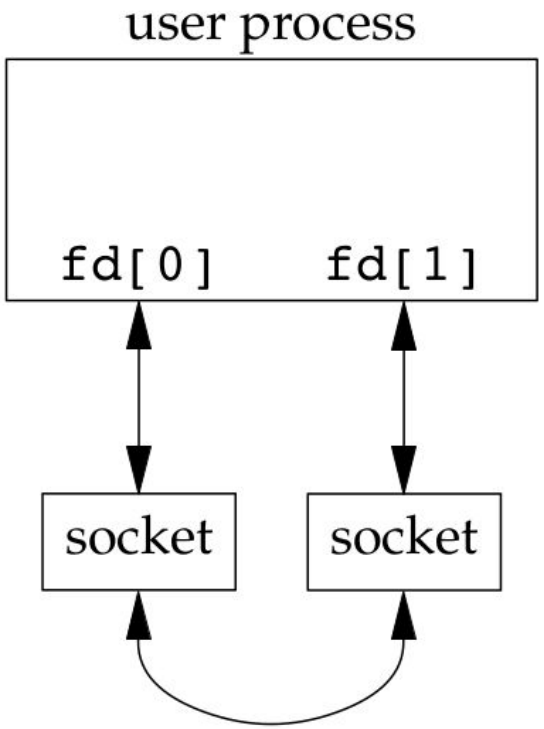
Tj. Jeśli zrobimy dwa razy zapisy po n bajtów, to odczyt $1.5*n$ bajtów zwróci *short count* równy n.

Ograniczony odpowiednik w WindowsNT!

Przenośna implementacja dwukierunkowego potoku

```c
#include <sys/socket.h>

/*
 * Returns a full-duplex pipe
 * (a UNIX domain socket) with
 * the two file descriptors
 * returned in fd[0] and fd[1].
 */
int fd_pipe(int fd[2])
{
  return socketpair(AF_UNIX, SOCK_STREAM, 0, fd);
}
```



Pytania?