Systemy operacyjne

(slajdy uzupełniające)

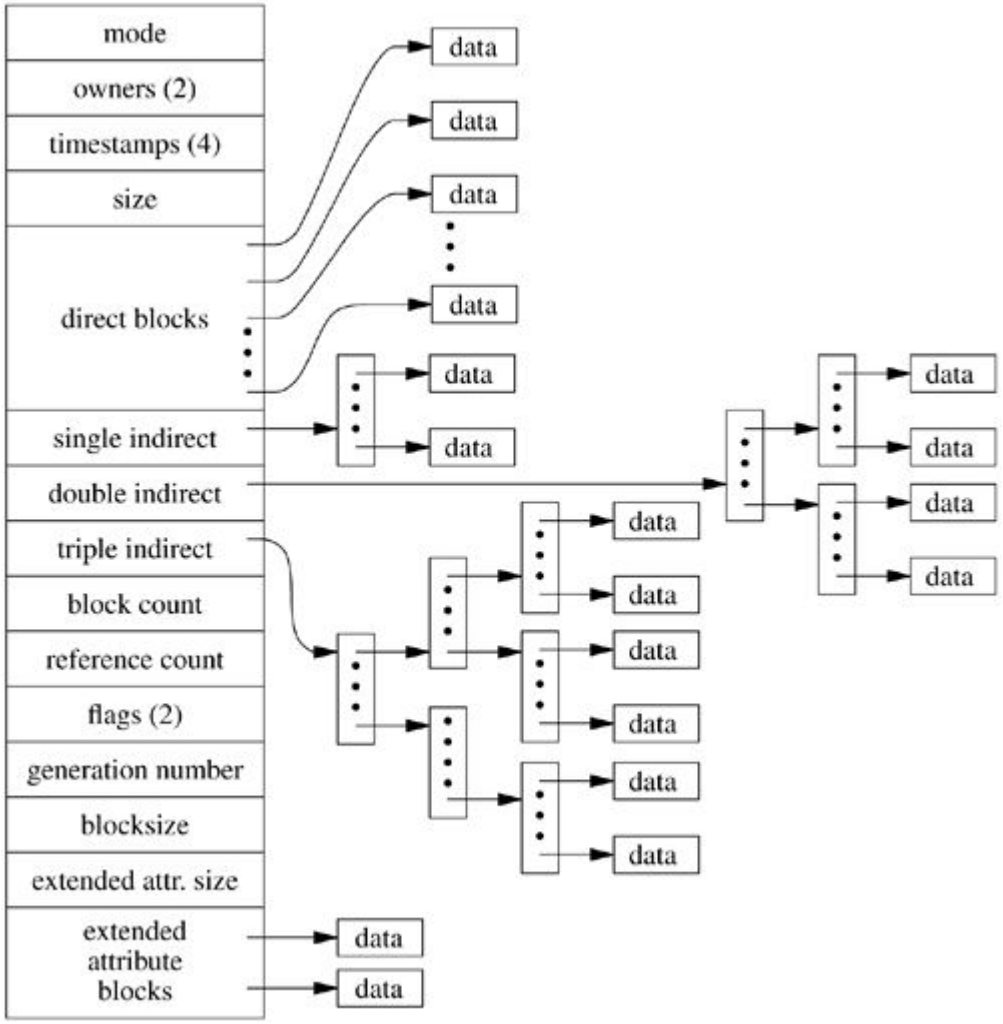
Wykład 5: Pliki (c.d.)

Pliki

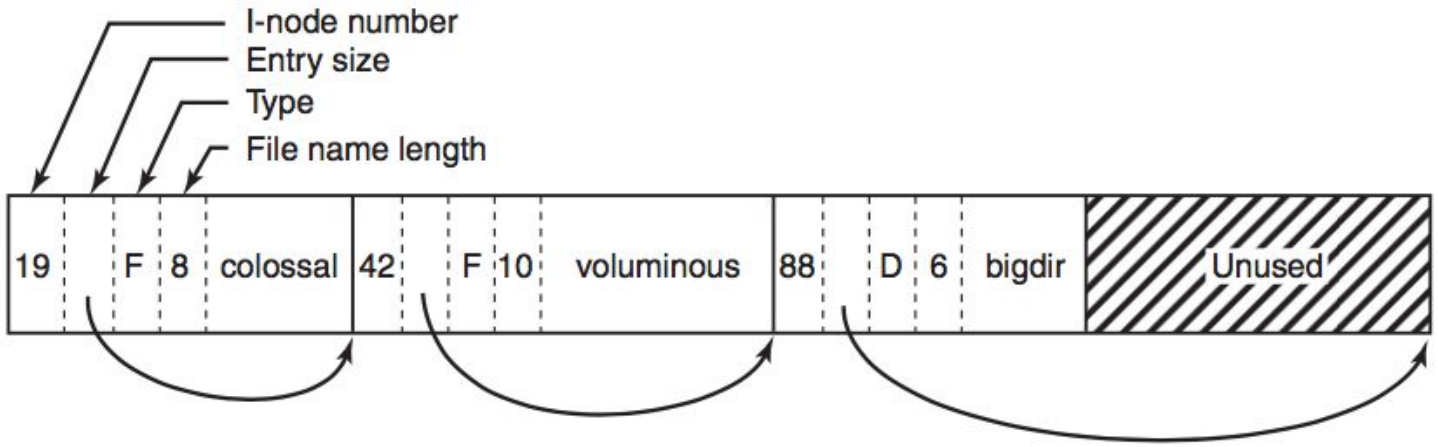
i-węzeł (ang. *i-node*)

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.

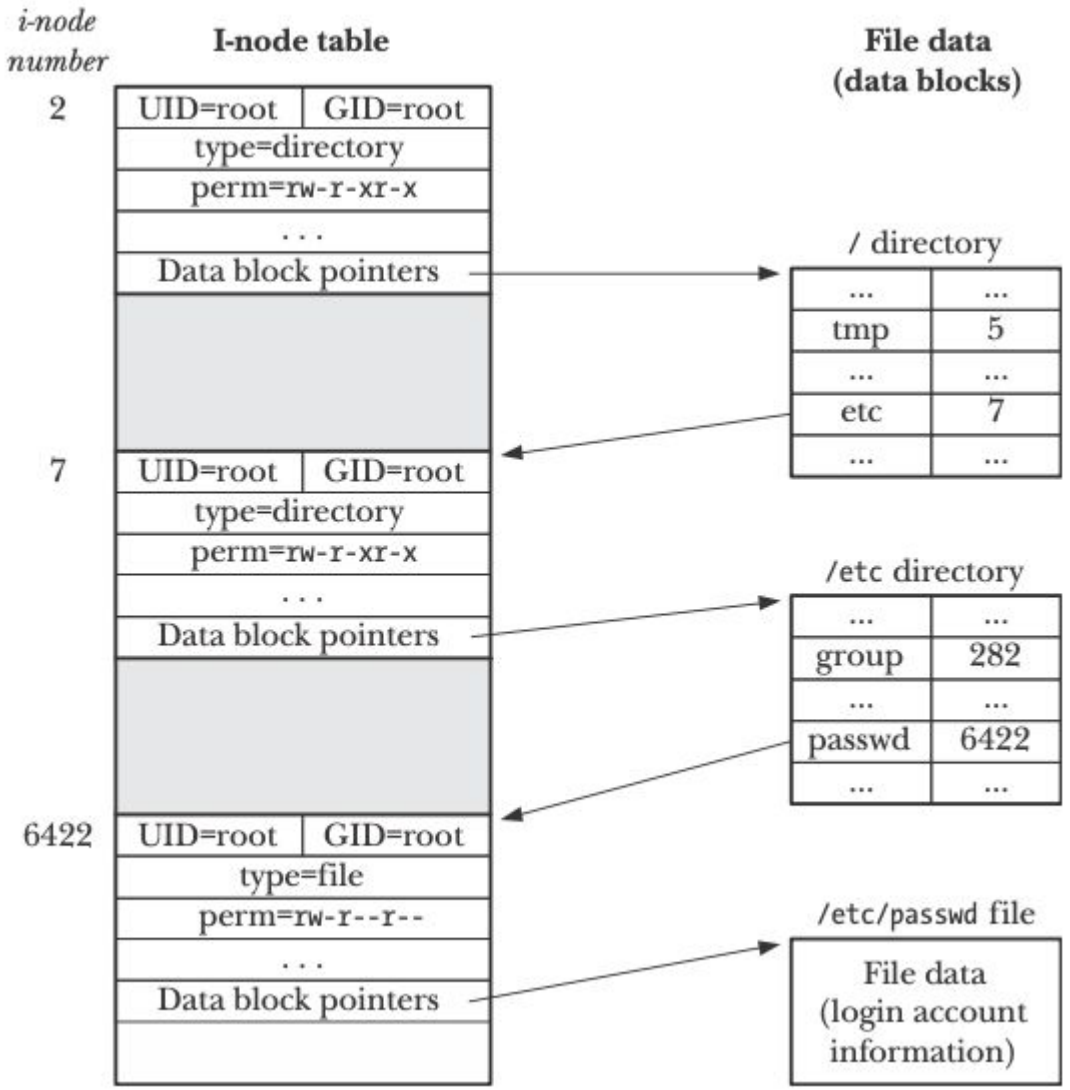
W wyniku operacji na katalogu w reprezentacji powstają nieużytki → rozmiar wpisu może być dużo większy niż nazwa pliku przechowywanego przez wpis. Co jakiś czas potrzebne **kompaktowanie**, które zmniejsza rozmiar katalogu i potencjalnie zwalnia nieużywane bloki na końcu.

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);
```

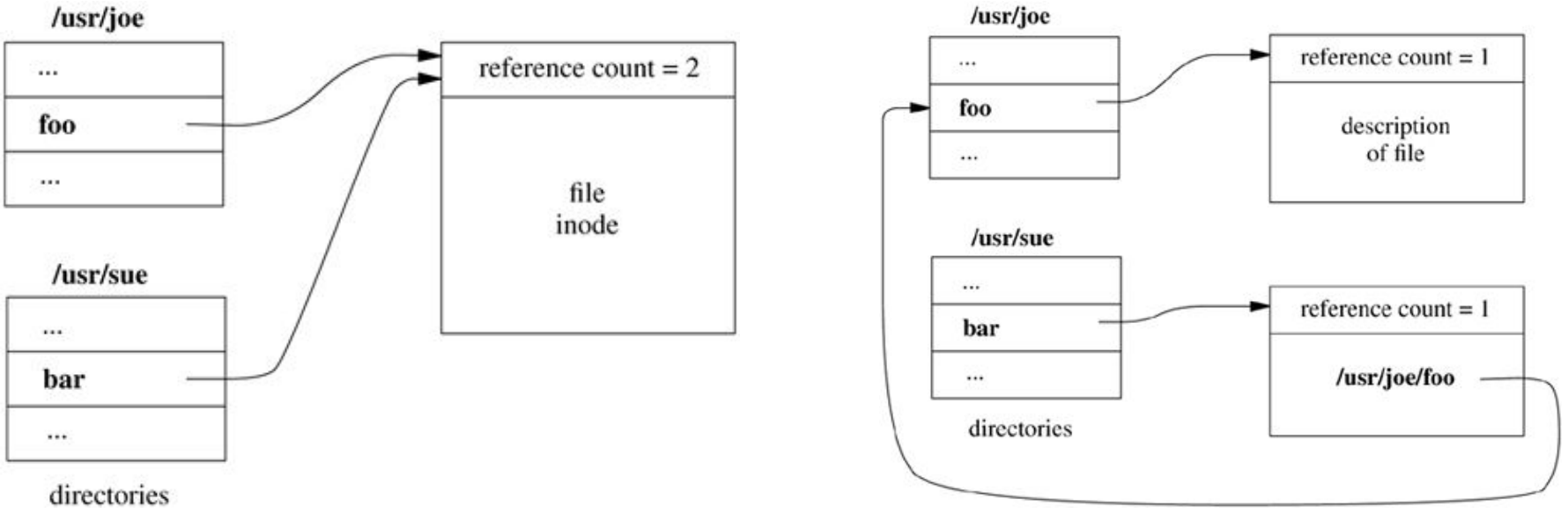
Dowiązania symboliczne (ang. *symbolic links*) specjalny typ pliku, który w zawartości przechowuje ścieżkę do innego pliku. System nie sprawdza poprawności tej ścieżki → może powstać pętla.

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



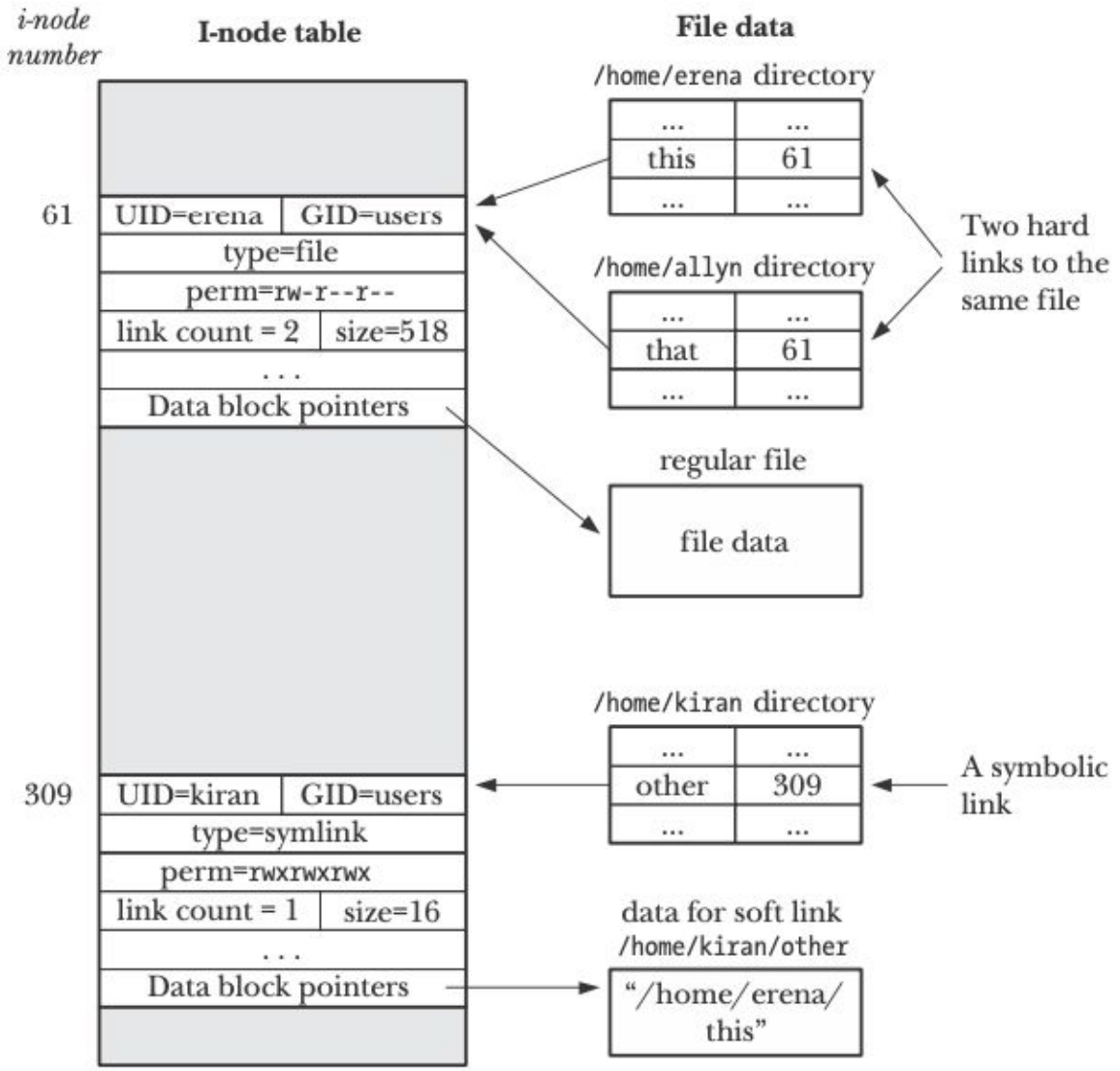
Dowiązania twarde to wskaźniki na i-węzły (licznik referencji!) plików → różne nazwy tego samego pliku w obrębie jednego systemu plików.

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 “**`/`**”).



Potoki

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

Potoki 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.

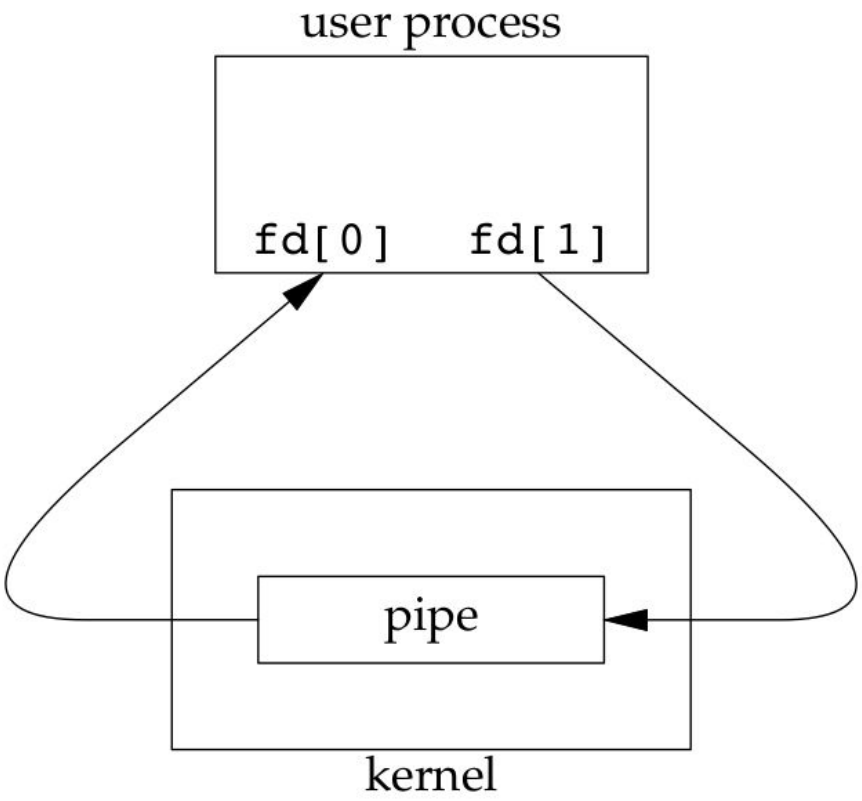
Nazwane potoki, tj. takie które posiadają nazwę w systemie plików, nazywamy FIFO (mkfifo).

Potoki występują również w WindowsNT.

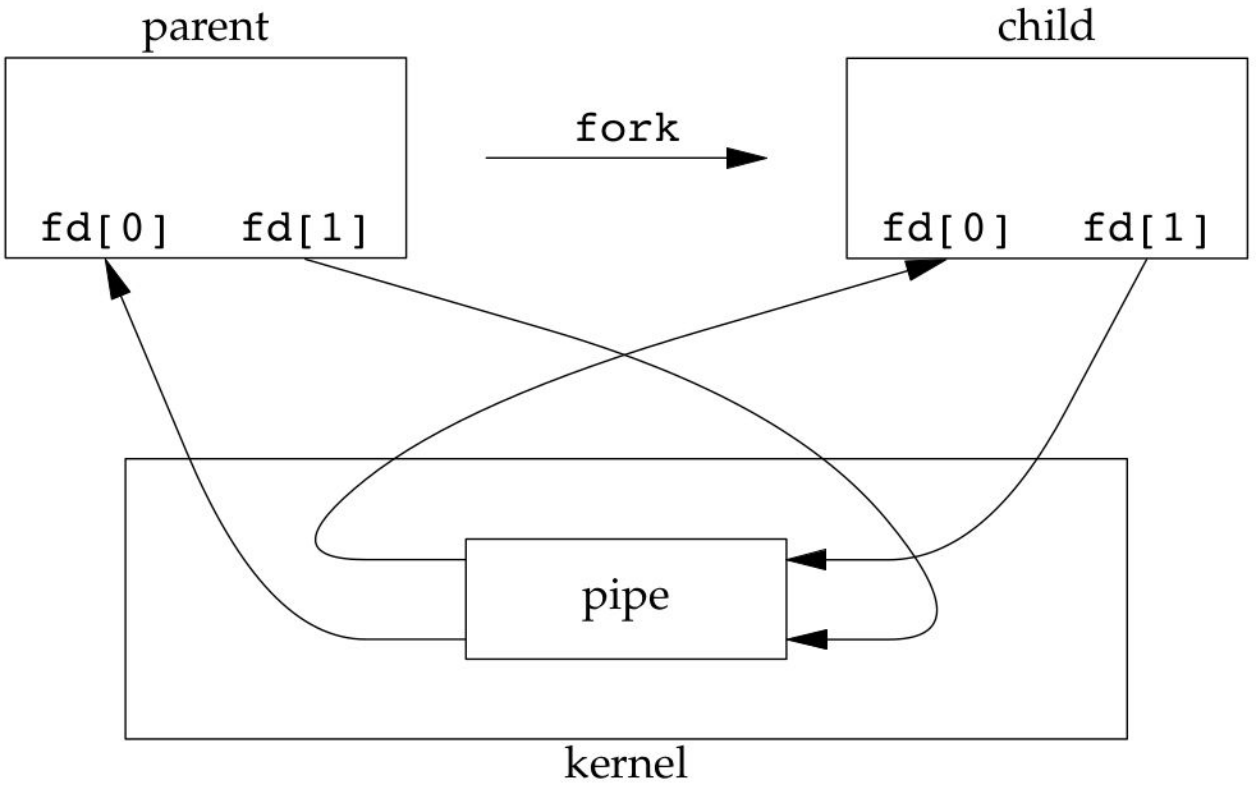
Potoki: przykład

```
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;
}
```

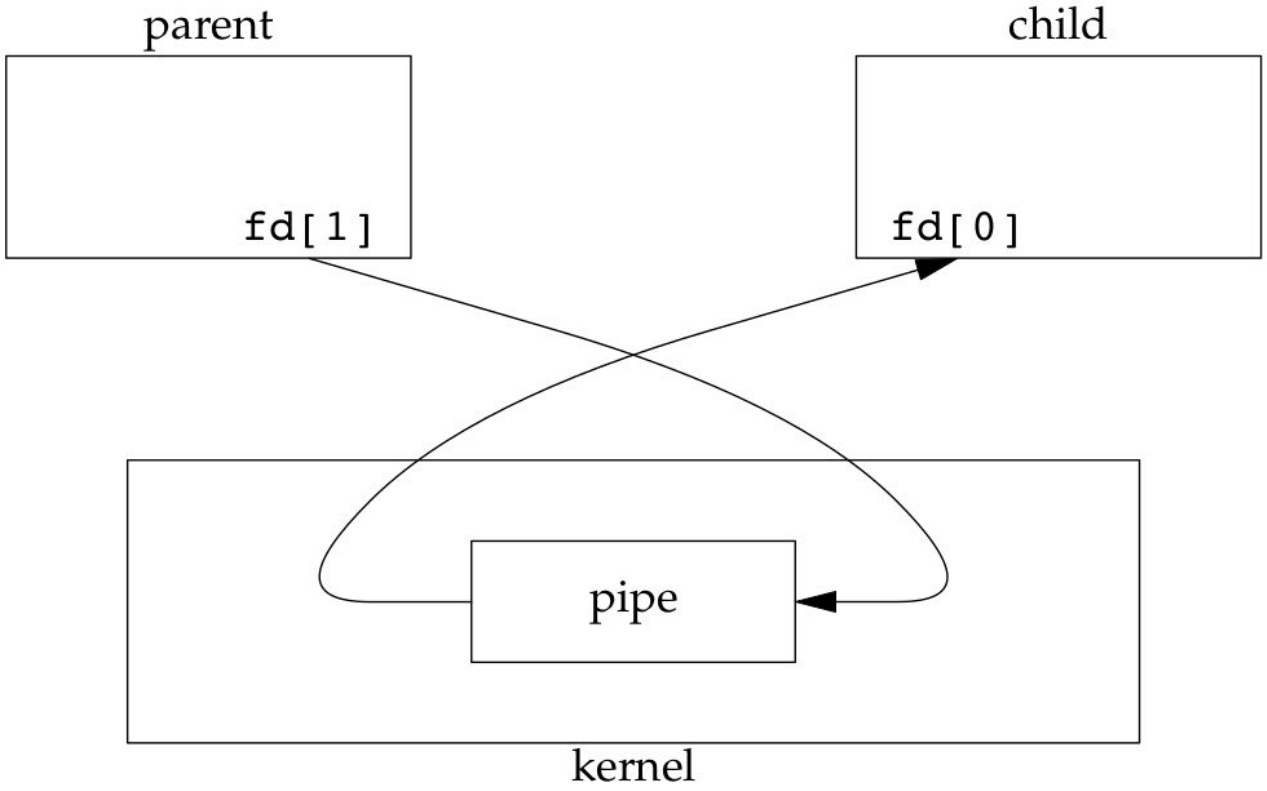
Etap 1: tworzenie potoku



Etap 2: wykonanie `fork()`



Etap 3: Zamknięcie niepotrzebnych końców



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).

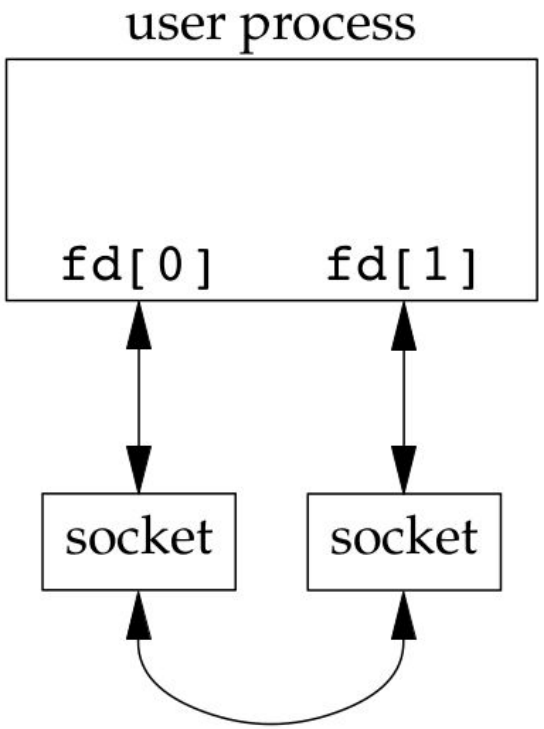
Tj. Jeśli zrobimy dwa razy zapisy po n bajtów, to odczyt $1.5*n$ bajtów zwróci *short count* równy n.

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);
}
```



Komunikaty pomocnicze

Dodatkowa funkcja gniazd domeny uniksowej → przesyłanie między procesami zasobów i tożsamości (`cmsg`).

`SCM_RIGHTS` duplikowanie i przesyłanie deskryptorów tj. otwartych plików, gniazd, potoków, semaforów, pamięci dzielonej, urządzeń, …

`SCM_CREDENTIALS` wysyłamy identyfikator procesu, numer użytkownika i grupy. Jądro weryfikuje tożsamość i dostarcza pakiet.

Przyład zastosowania: Tworzymy ciąg procesów, które będą wykonywały pewną akcję na otwartym zasobie. Jeśli zadanie się wykona, to przekazują zasób do następnego procesu.

Unix: operacje na deskryptorach

```
int flock(int fd, int operation);
```

```
int fcntl(int fd, int cmd, ...);
```

Blokady **doradcze** (ang. *advisory*) i **przymusowe** (ang. *mandatory*).
Te pierwsze przeważają w uniksach → Linux mandatory locking.
Blokady nie są wymuszane na programach, które ich nie używają!

Co i jak możemy blokować? Cały plik lub rekordy, do odczytu lub zapisu!
`fcntl` umożliwia zakładanie blokad i pieczęci, dzierżawienie plików, ustawianie flag dla otwartych plików oraz deskryptorów plików, itp.

Q: Czy blokada jest skojarzona: z plikiem, z otwartym plikiem, z procesem?
A: Blokady rekordów `fcntl` według `POSIX` z procesem. Jeśli proces umarł lub zamknął deskryptor odnoszący się do pliku → blokady usuwane.

Buforowanie

Koszt wywołań systemowych

Na podstawie *“The Linux Programming Interface”*:

1. Jądro Linux 2.6.30
2. System plików: **`ext2`**
3. Rozmiar bloku systemu plików: 4096 bajtów
4. Bufor w przestrzeni użytkownika: **`BUF_SIZE`** bajtów
5. Rozmiar pliku: ~100M bajtów

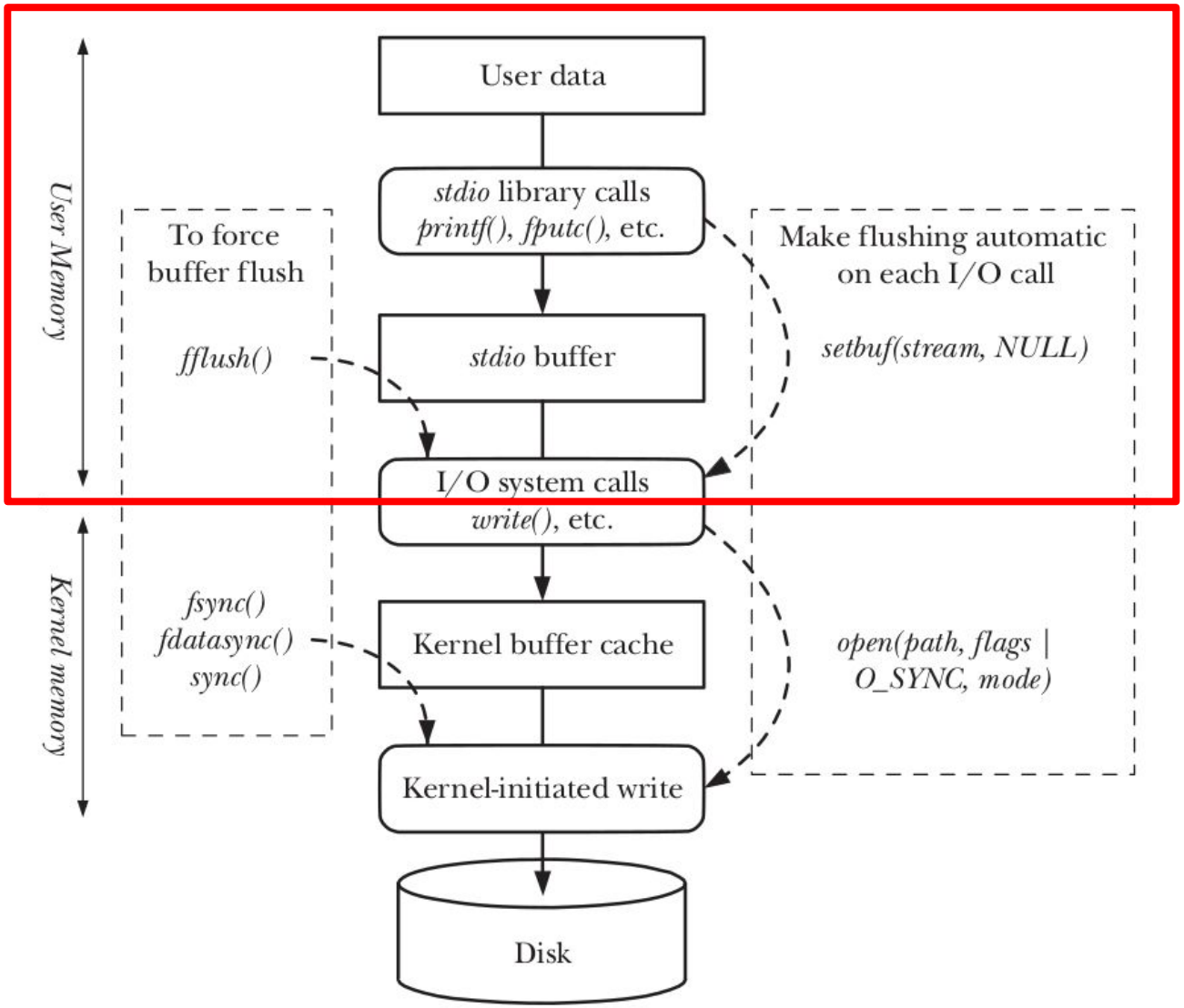
Kopiowanie pliku z użyciem `read(2)` i `write(2)`

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	0.32

Tworzenie zawartości pliku z użyciem `write(2)`

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	72.13	72.11	5.00	67.11
2	36.19	36.17	2.47	33.70
4	20.01	19.99	1.26	18.73
8	9.35	9.32	0.62	8.70
16	4.70	4.68	0.31	4.37
32	2.39	2.39	0.16	2.23
64	1.24	1.24	0.07	1.16
128	0.67	0.67	0.04	0.63
256	0.38	0.38	0.02	0.36
512	0.24	0.24	0.01	0.23
1024	0.17	0.17	0.01	0.16
4096	0.11	0.11	0.00	0.11
16384	0.10	0.10	0.00	0.10
65536	0.09	0.09	0.00	0.09

Buforowanie plików w przestrzeni użytkownika



Buforowanie biblioteki **`stdio`**

W strukturze **`FILE`** każdego strumienia jeden bufor.

```
void setbuf(FILE *stream, char *buf);

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

Function	*mode*	*buf*	Buffer and length	Type of buffering
`setbuf`		non-null	user *buf* of length `BUFSIZ`	fully buffered or line buffered
		`NULL`	(no buffer)	unbuffered
`setvbuf`	`_IOFBF`	non-null	user *buf* of length *size*	fully buffered
		`NULL`	system buffer of appropriate length	
	`_IOLBF`	non-null	user *buf* of length *size*	line buffered
		`NULL`	system buffer of appropriate length	
	`_IONBF`	(ignored)	(no buffer)	unbuffered

Domyślne tryb buforowania

Bufor domyślnie opróżniany w trakcie zamykania pliku **`fclose(3)`** otwartego do zapisu. W trakcie pracy możemy zawołać **`fflush(3)`**, żeby jawnie go opróżnić.

Dla plików dyskowych buforowanie pełne, dla plików terminala buforowanie liniami, dla **`stderr`** brak buforowania.

Jak sprawdzić czy plik jest terminalem?

```
int fileno(FILE *stream);
int isatty(int fd);
```

Domyślna wielkość bufora → **`st_blksize`** (**`statbuf`**).

Problemy z buforowaniem po stronie użytkownika

- Podwójne kopiowanie danych:
  - jądro kopiuje dane do bufora FILE,
  - użytkownik korzystając z funkcji **`stdio`** kopiuje dane do własnej pamięci.
- Utrata zawartości bufora:
  - zapisujemy dane z użyciem **`fprintf(3)`** albo **`fwrite(3)`**
  - zapominamy zawołać **`fclose(3)`**
  - wychodzimy z programu…
  - albo przychodzi sygnał, który kończy działanie programu.

Wywołania systemowe **`readv(2)`** i **`writev(2)`**

Motywacja: Piszemy nasz własny edytor tekstu.
Plik reprezentujemy w pamięci jako tablicę rekordów:

(długość linii, wskaźnik do zawartości linii)

Linie nie muszą być ułożone w pamięci sekwencyjnie!

Co musimy zrobić, żeby zapisać plik na dysk?

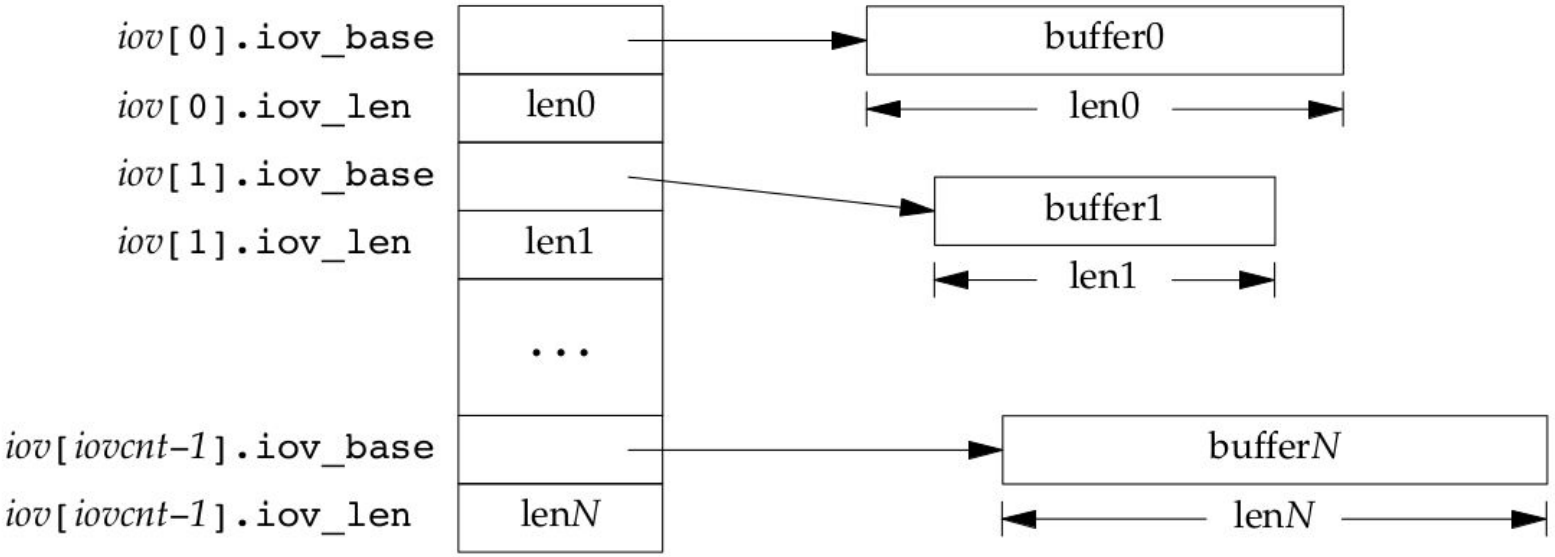
- Dla każdej linii zwołać write(2).
- Przygotować jeden wielki bufor, do którego wkopiujemy wszystkie linie i zapiszemy na dysk w jednym kroku.

Wywołania systemowe **`readv(2)`** i **`writev(2)`** c.d.

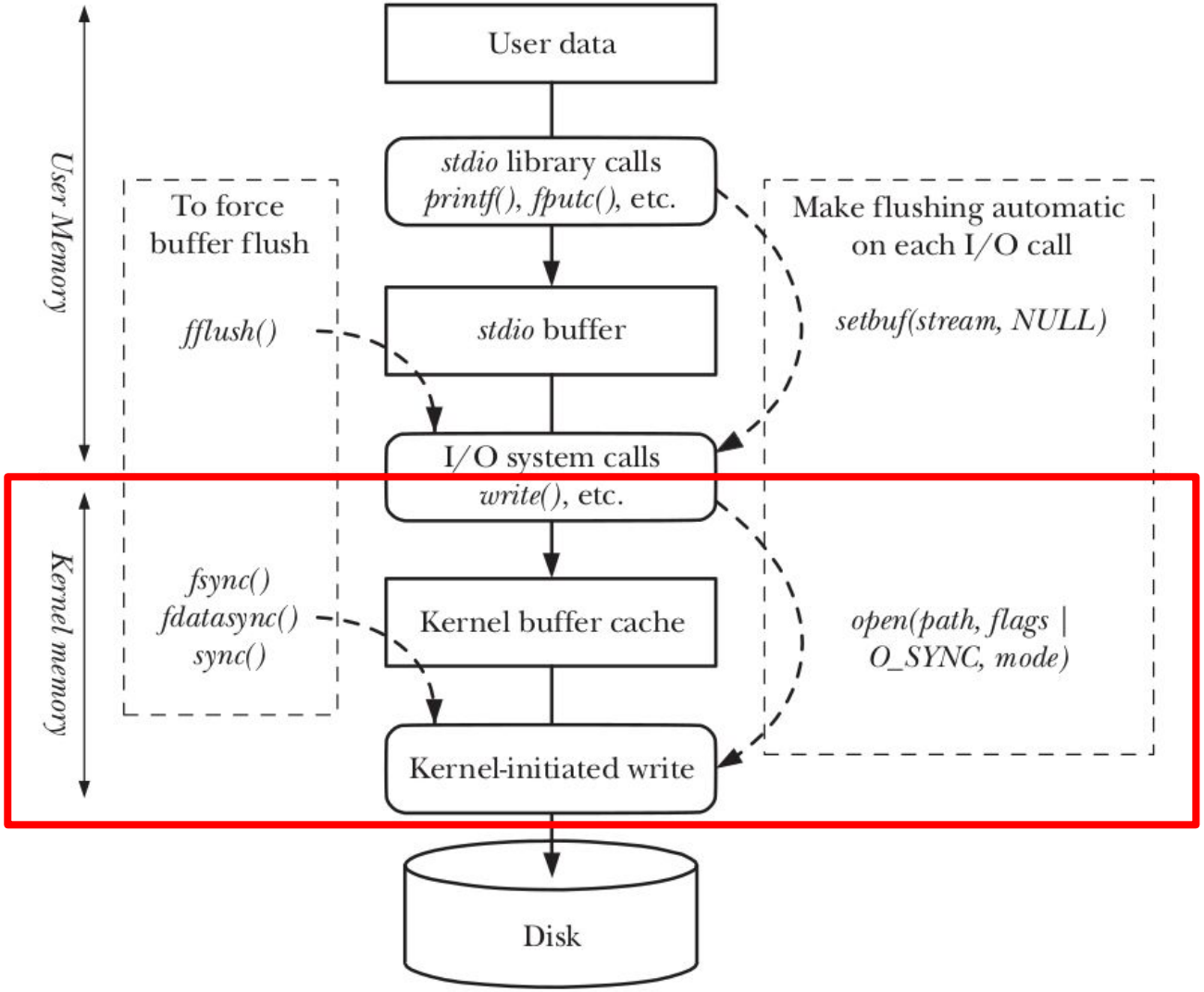
Rozwiązaniem *scatter read* i *gather write*:

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

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



Buforowanie plików w przestrzeni użytkownika



Pamięć podręczna buforów

Jądro utrzymuje sprowadzone z dysku bloki w ***buffer cache*** (termin stosowany zamiennie z ***page cache***).

Bufory te mogą być bezpośrednio odwzorowane w pamięci użytkownika (**`mmap`**), albo używane przez implementację wywołań systemowych **`read`** i **`write`**.

Zapis do pliku kończy się jedynie zapisem do bufora.
Po jakimś czasie jądro zapisze zawartość buforów na dysk.

Jądro wykorzystuje wolną pamięć RAM do cache'owania pamięci drugorzędnej (nośniki danych).

Problem z buforowaniem

Rozważmy serwer poczty przekazujący e-mail do serwera B:

1. Odbiera e-mail z serwera A
2. Zapisuje go na dysku
3. Wysyła potwierdzenie (odebrałem!) do serwera A
4. Serwer A kasuje wiadomość z dysku
5. Serwer B przekazuje e-mail dalej

Jeśli serwer B ulegnie awarii (np. brak prądu) po wykonaniu punktu 3, to czy e-mail będzie bezpieczny?

Nie! Zawartość pliku z wiadomością mogła być nadal przechowywana w buforze systemu plików (RAM).

Buforowanie danych i metadanych

`sync(2)` synchronizuje wszystkie bufory jądra z pamięcią drugorzędną

`fsync(2)` synchronizuje dane i metadane wybranego pliku

`fdatasync(2)` synchronizuje tylko dane pliku

Jaka jest różnica? Dopisujemy na koniec pliku – dane zostały wypisane na dysk, a rozmiar pliku nie, bo jest w metadanych!

Dodatkowo w trakcie otwierania pliku możemy przekazać do **`open(2)`** flagi **`O_SYNC`** i **`O_DSYNC`**, które mają taki sam efekt co **`fsync`** i **`fdatasync`** przy każdej operacji **`write`**.

Pytanie: Czy **`*sync`** zapewnia spójności danych na dysku?

Pytania?