Systemy operacyjne

(slajdy uzupełniające)

Wykład 5: Pliki, katalogi i potoki

Pliki i katalogi

Dowiązania symboliczne

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

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

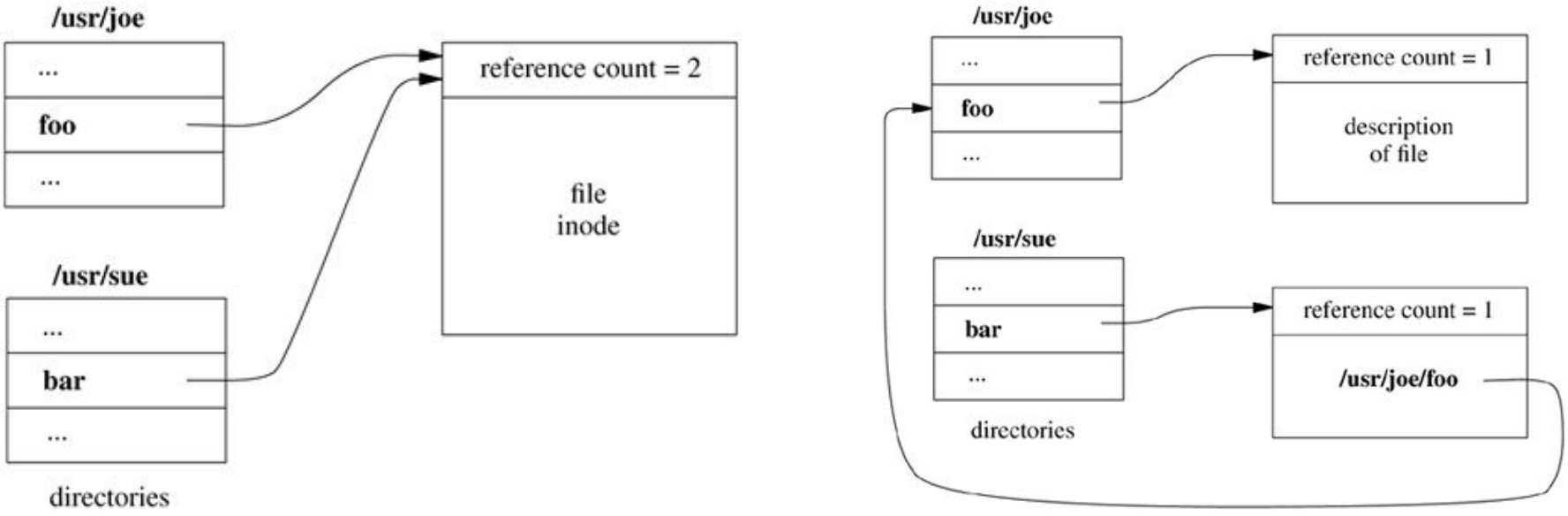
Dowiązania symboliczne (ang. *symbolic links*) specjalny typ pliku, który w zawartości przechowuje ścieżkę do innego pliku. System nie sprawdza poprawności tej ścieżki → może powstać pętla.

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`.

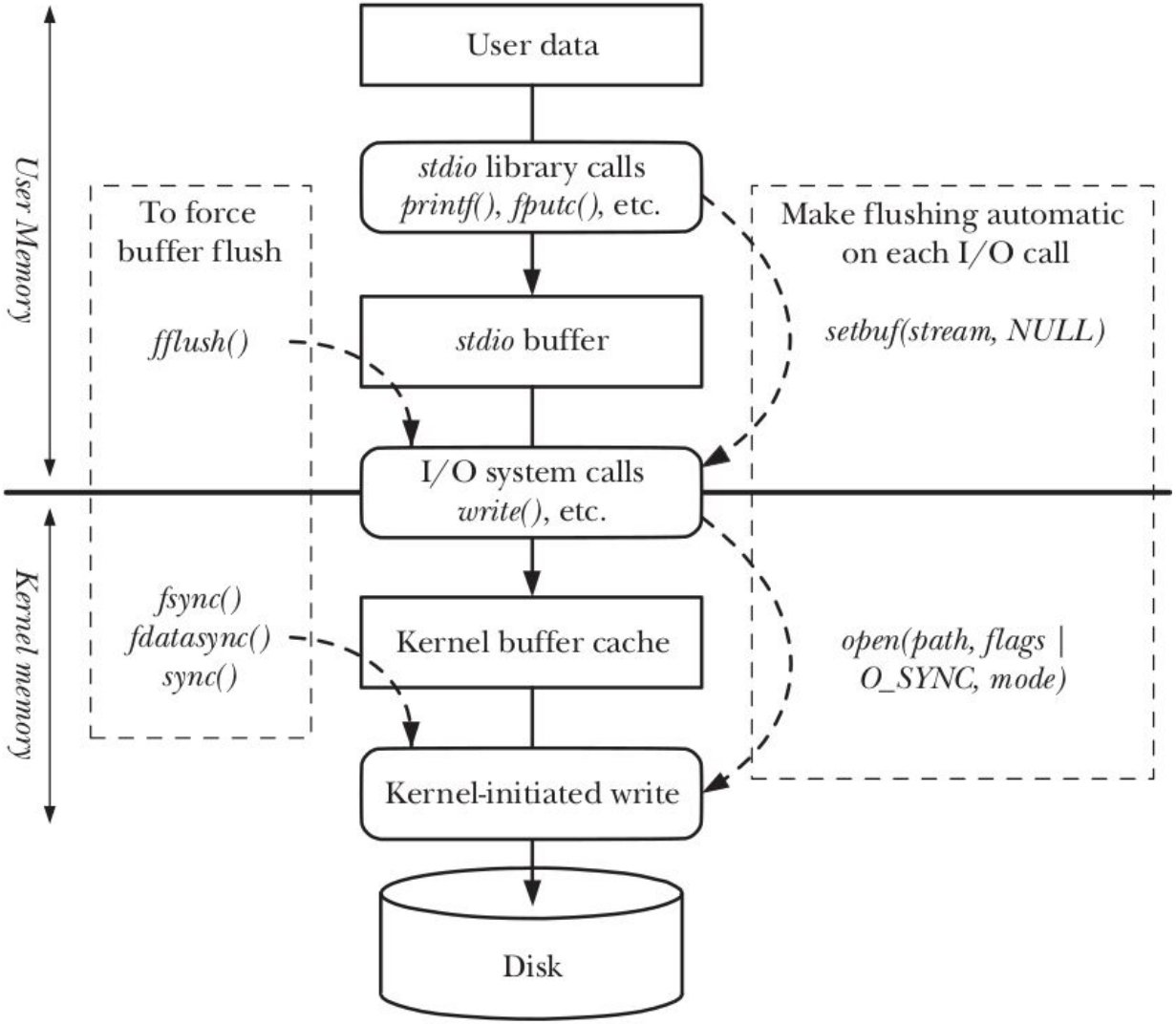
Dowiązanie symboliczne vs. twarde



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.

Buforowanie plików



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` synchronizuje wszystkie bufory jądra z pamięcią drugorzędną

`fsync` (**`O_SYNC`**) synchronizuje dane i metadane wybranego pliku

`fdatasync` (**`O_DSYNC`**) synchronizuje tylko dane pliku

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

Q: Czy zawsze chcemy synchronizować jednocześnie dane i metadane?
A: Czas ostatniego dostępu (ang. *access time*) pewnie nie, szczególnie dla dysków półprzewodnikowych.

Unix: operacje na deskryptorach

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

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

Blokady **doradcze** (ang. *advisory*) i **przymusowe** (ang. *mandatory*).
W uniksach te pierwsze wystepują częściej → Linux mandatory locking.

Co i jak możemy blokować? Cały plik lub rekordy, do odczytu lub zapisu!
`fcntl` umożliwia zakładanie blokad i pieczęci, dzierżawienie 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.

Wektorowe wejście-wyjście

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

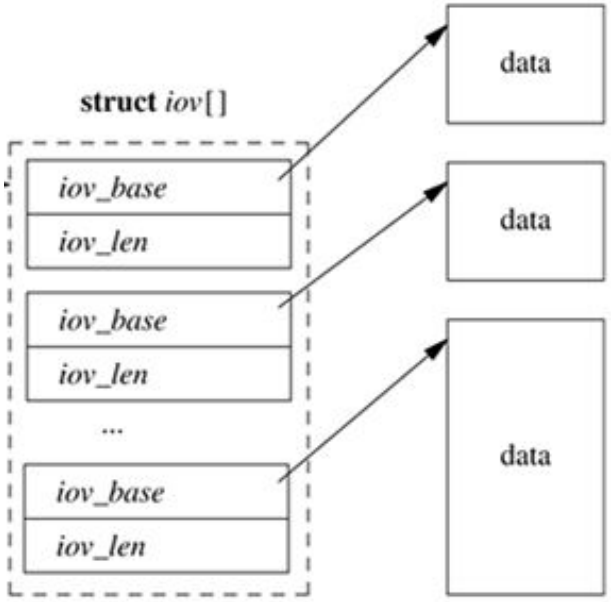
```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
```

Motywacja: Aktualizujemy rekordy bazy danych, każdy po kilkadziesiąt bajtów. Rekordy są rozrzucone po całej tabeli.

Zapis rekordu wymaga dwóch operacji: `seek`, `write`. Koszt wywołania systemowego jest nie do zignorowania…

Jak zminimalizować koszt aktualizacji?

Scatter-gather I/O!



Potoki i gniazda

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.

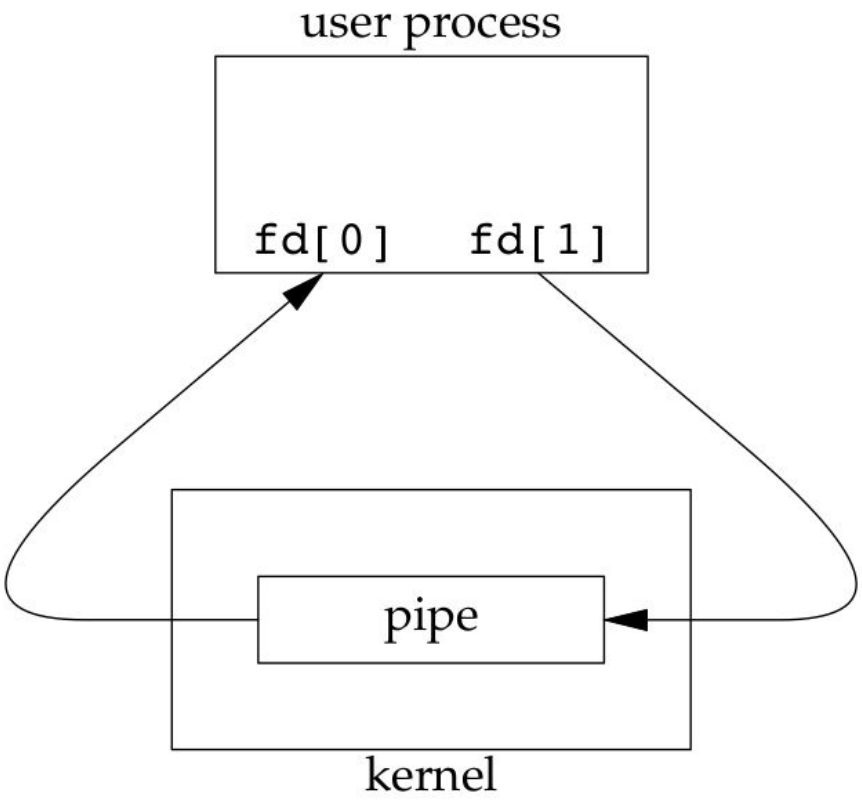
Nazwane potoki, tj. takie które posiadają nazwę w systemie plików, nazywamy FIFO (mkfifo).

Potoki występują również w WindowsNT.

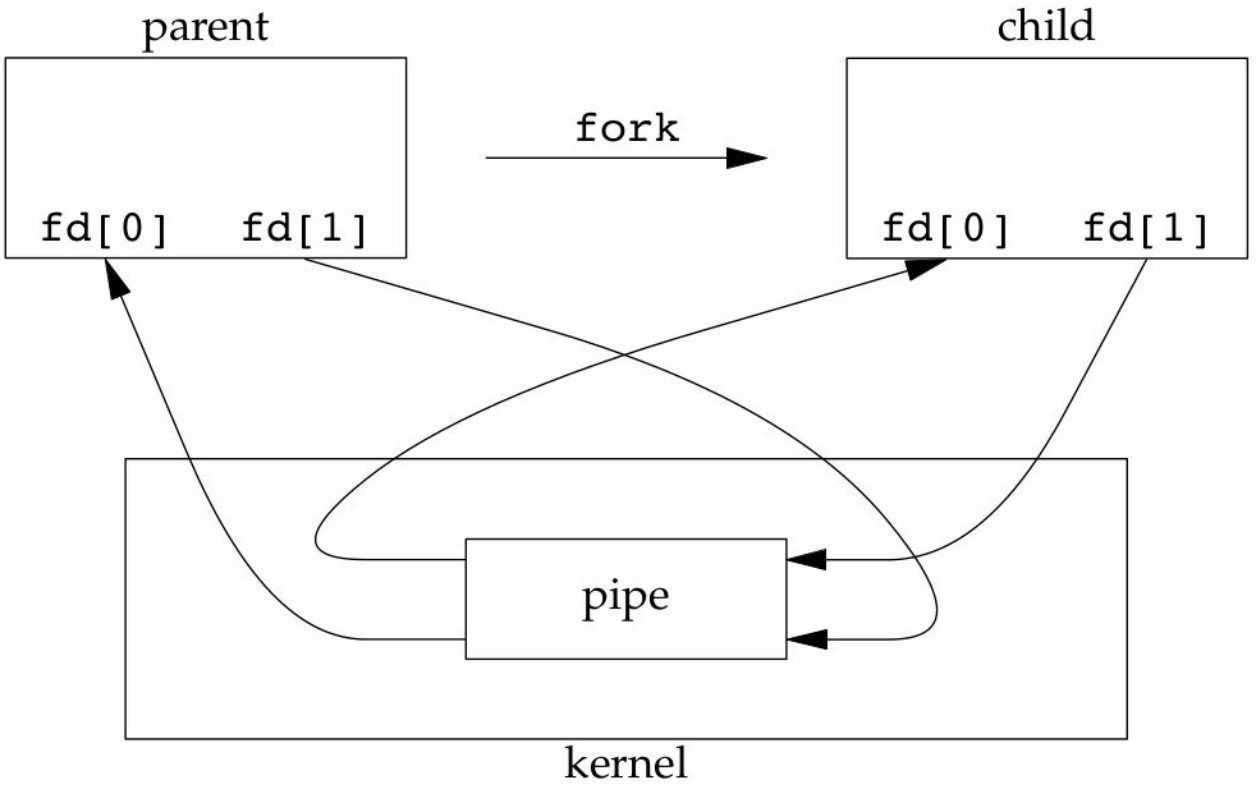
Potoki: przykład

```
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;
}
```

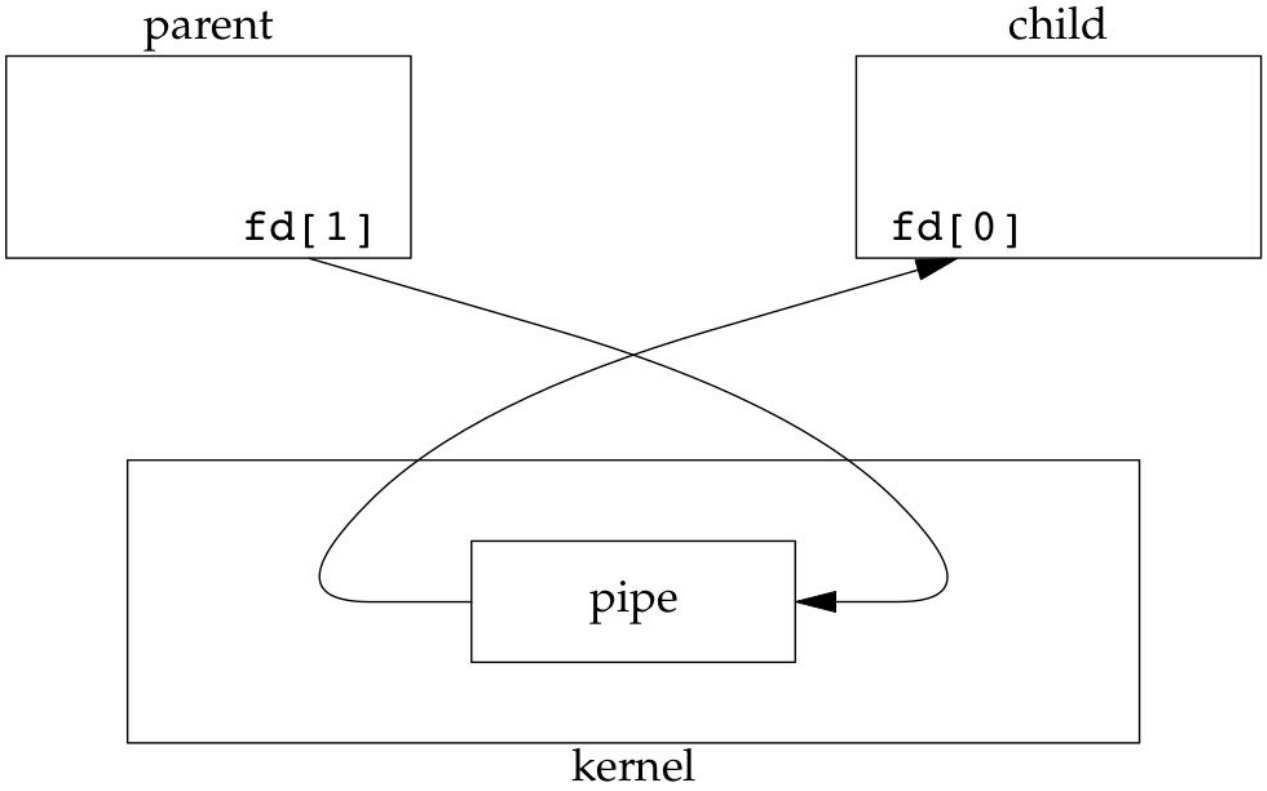
Etap 1: tworzenie potoku



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 pakietami).

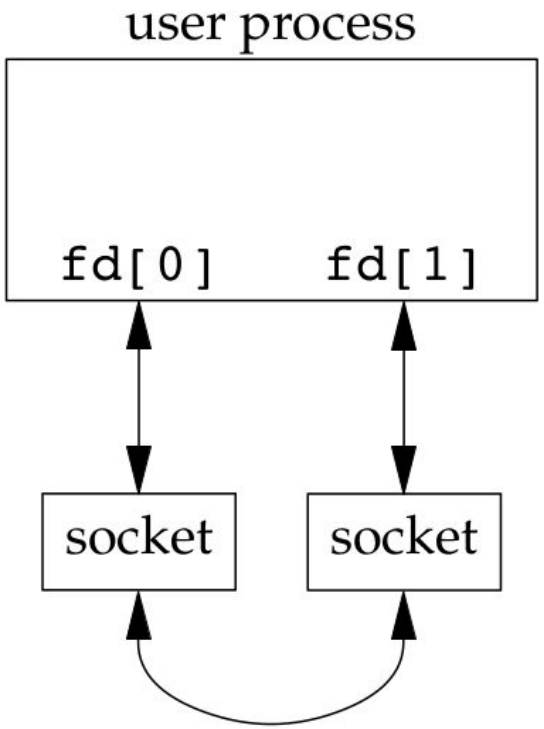
Tj. Jeśli zrobimy dwa razy zapisy po n bajtów, to odczyt $1.5*n$ bajtów zwróci *short count* równy n.

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.

Pytania?