4 Łącza

Łącza w systemie UNIX są plikami specjalnymi, służącymi do komunikacji pomiędzy procesami. Łącza mają kilka cech typowych dla plików zwykłych, czyli posiadają swój i-węzeł, posiadają bloki z danymi (chociaż ograniczoną ich liczbę), na otwartych łączach można wykonywać operacje zapisu i odczytu. Łącza od plików zwykłych odróżniają następujące cechy:

- ograniczona liczba bloków — łącza mają rozmiar 4KB – 8KB w zależności od konkretnego systemu,
- dostęp sekwencyjny — na łączach można wykonywać tylko operacje zapisu i odczytu, nie można natomiast przemieszczać wskaźnika bieżącej pozycji (nie można wykonywać funkcji `lseek`),
- sposób wykonywania operacji zapisu i odczytu — dane odczytywane z łącza są zarazem usuwane (nie można ich odczytać ponownie),
- proces jest blokowany w funkcji `read` na pustym łączu, jeśli jest otwarty jakiś deskryptor tego łącza do zapisu,
- proces jest blokowany w funkcji `write`, jeśli w łączu nie ma wystarczającej ilości wolnego miejsca, żeby zmieścić zapisywany blok[2].

Jak już wspomniano przy opisie funkcji systemowej `read` (str. 7), zwrócenie przez nią wartości 0 jest wskazaniem osiągnięcia końca pliku. Koniec odczytu danych z łącza następuje dopiero, gdy wszystkie dane zostaną odczytane i **wszystkie deskryptory do zapisu zostaną zamknięte**. Pozostawienie otwartego deskryptora do zapisu jest interpretowane przez system jako możliwość dalszego przekazywania kolejnych danych, czego konsekwencją jest blokowanie procesów odczytujących w funkcji `read`. Sytuacja taka może prowadzić do zakleszczenia (ang. deadlock). Istotne jest zatem zamykanie zbędnych deskryptorów, zwłaszcza deskryptorów łącza do zapisu.

W systemie UNIX wyróżnia się dwa rodzaje łączy — *łącza nazwane* i *łącza nienazwane*. Zwyczajowo przyjęło się określać łącza nazwane terminem *kolejki FIFO*, a łącza nienazwane terminem *potoki*. Łącze nazwane ma dowiązanie w systemie plików, co oznacza, że jego nazwa jest widoczna w jakimś katalogu i może ona służyć do identyfikacji łącza. Łącze nienazwane nie ma nazwy w żadnym katalogu i istnieje tak długo po utworzeniu, jak długo otwarty jest jakiś deskryptor tego łącza.

4.1 Sposób korzystania z łącza nienazwanego

Ponieważ łącze nienazwane nie ma dowiązania w systemie plików, nie można go identyfikować przez nazwę. Jeśli procesy chcą się komunikować za pomocą takiego łącza, muszą znać jego deskryptory. Oznacza to, że procesy muszą uzyskać deskryptory tego samego łącza, nie znając jego nazwy. Jedynym sposobem przekazania informacji o łączu nienazwanym jest przekazanie jego deskryptorów procesom potomnym dzięki dziedziczeniu tablicy otwartych plików od swojego procesu macierzystego. Za pomocą łącza nienazwanego mogą się zatem komunikować procesy, z których jeden otworzył łącze nienazwane, a następnie utworzył pozostałe komunikujące się procesy, które w ten sposób otrzymają w tablicy otwartych plików deskryptory istniejącego łącza.

Łącze nienazwane tworzy funkcja systemowa `pipe`, zwracając 2 deskryptory — jeden do odczytu danych z łącza, a drugi do zapisu danych do łącza.

[2] Wyjątkiem od tej zasady jest przypadek, w którym łącze funkcjonuje w trybie bez blokowania (jest ustawiona flaga `O_NDELAY`).

`int pipe(int fd[2])` — utworzenie łącza nienazwanego. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.

Opis parametrów:

fd tablica 2 deskryptorów, która jest parametrem wyjściowym (`fd[0]` jest deskryptorem potoku do odczytu, a `fd[1]` jest deskryptorem potoku do zapisu).

Listing 1 pokazuje przykładowe użycie łącza do przekazania napisu (ciągu znaków) `Hallo!` z procesu potomnego do macierzystego.

Listing 1: Przykład użycia łącza nienazwanego w komunikacji przodek-potomek

```
main() {
   int pdesk[2];

   if (pipe(pdesk) == -1){
      perror("Tworzenie potoku");
      exit(1);
   }

   switch(fork()){
      case -1:    // blad w tworzeniu procesu
         perror("Tworzenie procesu");
         exit(1);
      case 0:     // proces potomny
         if (write(pdesk[1], "Hallo!", 7) == -1){
            perror("Zapis do potoku");
            exit(1);
         }
         exit(0);
      default: {  // proces macierzysty
         char buf[10];
         if (read(pdesk[0], buf, 10) == -1){
            perror("Odczyt z potoku");
            exit(1);
         }
         printf("Odczytano z potoku: %s\n", buf);
      }
   }
}
```

Opis programu: Do utworzenia i zarazem otwarcia łącza nienazwanego służy funkcja systemowa `pipe`, wywołana przez proces macierzysty (linia 4). Następnie tworzony jest proces potomny przez wywołanie funkcji systemowej `fork` w linii 9, który dziedziczy tablicę otwartych plików swojego przodka. Warto zwrócić uwagę na sposób sprawdzania poprawności wykonania funkcji systemowych zwłaszcza w przypadku funkcji `fork`, która kończy się w dwóch procesach — macierzystym i potomnym. Proces potomny wykonuje program zawarty w liniach 14–19 i zapisuje do potoku ciąg 7 bajtów spod adresu początkowego napisu `Hallo!`. Zapis tego ciągu polega na wywołaniu funkcji systemowej `write` na odpowiednim deskryptorze, podobnie jak w przypadku pliku zwykłego.

Proces macierzysty (linie 20–25) próbuje za pomocą funkcji `read` na odpowiednim deskryptorze odczytać ciąg 10 bajtów i umieścić go w buforze wskazywanym przez `buf` (linia 21). `buf` jest adresem początkowym tablicy znaków, zadeklarowanej w linii 20. Odczytany ciąg znaków może być krótszy, niż to wynika z rozmiaru bufora i wartości trzeciego parametru funkcji `read` (odczytane zostanie mniej niż 10 bajtów). Zawartość bufora, odczytana z potoku, wraz z odpowiednim napisem zostanie przekazana na standardowe wyjście.

Listing 2 zawiera zmodyfikowaną wersję przykładu przedstawionego na listingu 1. W poniższym przykładzie zakłada się, że wszystkie funkcje systemowe wykonują się poprawnie, w związku z czym w kodzie programu nie ma reakcji na błędy.

Listing 2: Przykład odczytu z pustego łącza

```
1 main() {
    int pdesk[2];

4   pipe(pdesk);

    if (fork() == 0){ // proces potomny
7      write(pdesk[1], "Hallo!", 7);
       exit(0);
    }
10  else { // proces macierzysty
       char buf[10];
       read(pdesk[0], buf, 10);
13     read(pdesk[0], buf, 10);
       printf("Odczytano z potoku: %s\n", buf);
    }
16 }
```

Opis programu: Podobnie, jak w przykładzie na listingu 1, proces potomny przekazuje macierzystemu przez potok ciąg znaków `Hallo!`, ale proces macierzysty próbuje wykonać dwa razy odczyt zawartości tego potoku. Pierwszy odczyt (linia 12) będzie miał taki sam skutek jak w poprzednim przykładzie. Drugi odczyt (linia 13) spowoduje zawieszenie procesu, gdyż potok jest pusty, a proces macierzysty ma otwarty deskryptor do zapisu. Gdyby się tak zdarzyło, że proces macierzysty nie odczyta całego bloku zapisanego przez potomka — co teoretycznie jest możliwe — drugie wywołanie funkcji `read` nie zablokuje procesu macierzystego, lecz zwróci kolejną porcję danych zapisanych przez potomka.

Listing 3 pokazuje sposób przejęcia wyniku wykonania standardowego programu systemu UNIX (w tym przypadku `ls`) w celu dalszego przetworzenia (w tym przypadku konwersji małych liter na duże). Pobranie argumentów z linii poleceń umożliwia przekazanie ich do programu wykonywanego przez proces potomny.

Listing 3: Konwersja wyniku polecenia `ls`

```
1 #define MAX 512

  main(int argc, char* argv[]) {
4   int pdesk[2];

    if (pipe(pdesk) == -1){
7      perror("Tworzenie potoku");
       exit(1);
    }
10
    switch(fork()){
       case -1: // blad w tworzeniu procesu
13        perror("Tworzenie procesu");
          exit(1);
       case 0: // proces potomny
16        dup2(pdesk[1], 1);
          execvp("ls", argv);
          perror("Uruchomienie programu ls");
19        exit(1);
       default: { // proces macierzysty
          char buf[MAX];
22        int lb, i;

          close(pdesk[1]);
25        while ((lb=read(pdesk[0], buf, MAX)) > 0){
```

```
            for(i=0; i<lb; i++)
               buf[i] = toupper(buf[i]);
28          if (write(1, buf, lb) == -1){
               perror ("Zapis na standardowe wyjscie");
               exit(1);
31          }
         }
         if (lb == -1){
34          perror("Odczyt z potoku");
            exit(1);
         }
37    }
   }
}
```

Opis programu: Program jest podobny do przykładu z listingu 1, przy czym w procesie potomnym następuje przekierowanie standardowego wyjścia do potoku (linia 16), a następnie uruchamiany jest program `ls` (linia 17). W procesie macierzystym dane z potoku są sukcesywnie odczytywane (linia 25), małe litery w odczytanym bloku konwertowane są na duże (linie 26–27), a następnie blok jest zapisywany na standardowym wyjściu procesu macierzystego. Powyższa sekwencja powtarza się w pętli (linie 25–32) tak długo, aż funkcja systemowa `read` zwróci wartość 0 (lub –1 w przypadku błędu). Istotne jest zamknięcie deskryptora potoku do zapisu (linia 24) w celu uniknięcia zawieszenia procesu macierzystego w funkcji `read`.

Przykład na listingu 4 pokazuje realizację programową potoku `ls | tr a-z A-Z`, w którym proces potomny wykonuje polecenie `ls`, a proces macierzysty wykonuje polecenie `tr`. Funkcjonalnie jest to odpowiednik programu z listingu 3.

Listing 4: Programowa realizacja potoku `ls | tr a-z A-Z` na łączu nienazwanym

```
main(int argc, char* argv[]) {
2   int pdesk[2];

    if (pipe(pdesk) == -1){
5      perror("Tworzenie potoku");
       exit(1);
    }
8
    switch(fork()){
       case -1: // blad w tworzeniu procesu
11        perror("Tworzenie procesu");
          exit(1);
       case 0: // proces potomny
14        dup2(pdesk[1], 1);
          execvp("ls", argv);
          perror("Uruchomienie programu ls");
17        exit(1);
       default: { // proces macierzysty
          close(pdesk[1]);
20        dup2(pdesk[0], 0);
          execlp("tr", "tr", "a-z", "A-Z", 0);
          perror("Uruchomienie programu tr");
23        exit(1);
       }
    }
26 }
```

Opis programu: Program procesu potomnego (linie 16–19) jest taki sam, jak w przykładzie na listingu 3. W procesie macierzystym następuje z kolei przekierowanie standardowego wejścia na pobieranie danych z potoku (linia 22), po czym następuje uruchomienie programu `tr` (linia 23). W celu zagwarantowania, że przetwarzanie zakończy się w sposób naturalny konieczne jest zamknięcie wszystkich deskryptorów potoku do zapisu. Deskryptory w procesie potomnym zostaną zamknięte wraz z jego zakończeniem, a deskryptor w procesie macierzystym zamykany jest w linii 21.

4.2 Sposób korzystania z łącza nazwanego

Operacje zapisu i odczytu na łączu nazwanym wykonuje się tak samo, jak na łączu nienazwanym, inaczej natomiast się je tworzy i otwiera. Łącze nazwane tworzy się poprzez wywołanie funkcji `mkfifo` w programie procesu lub przez wydanie polecenia `mkfifo` na terminalu. Funkcja `mkfifo` tworzy plik specjalny typu łącze podobnie, jak funkcja `creat` tworzy plik zwykły. Funkcja `mkfifo` nie otwiera jednak łącza i tym samym nie przydziela deskryptorów. Łącze nazwane otwierane jest funkcją `open` podobnie jak plik zwykły, przy czym łącze musi zostać otwarte jednocześnie w trybie do zapisu przez jeden proces i w trybie do odczytu przez inny proces. W przypadku wywołania funkcji `open` tylko w jednym z tych trybów proces zostanie zablokowany aż do momentu, gdy inny proces nie wywoła funkcji `open` w trybie komplementarnym.

`int mkfifo(const char *`*pathname*`, mode_t` *mode*`)` — utworzenie pliku specjalnego typu łącze. Funkcja zwraca 0 w przypadku poprawnego zakończenia lub –1 w przypadku błędu.
Opis parametrów:
pathname nazwa pliku (w szczególności nazwa ścieżkowa),
mode prawa dostępu do nowo tworzonego pliku.

Program na listingu 5 pokazuje przykładowe tworzenie łącza i próbę jego otwarcia w trybie do odczytu.

Listing 5: Przykład tworzenie i otwierania łącza nazwanego

```
#include <fcntl.h>

main(){
    mkfifo("kolFIFO", 0600);
    open("kolFIFO", O_RDONLY);
}
```

Opis programu: Funkcja `mkfifo` (linia 4) tworzy plik specjalny typu łącze o nazwie `kolFIFO` z prawem zapisu i odczytu dla właściciela. W linii 5 następuje próba otwarcia łącza w trybie do odczytu. Proces zostanie zawieszony w funkcji `open` do czasu, aż inny proces będzie próbował otworzyć tę samą kolejkę w trybie do zapisu.

Listing 6 pokazuje realizację przykładu z listingu 1, w której wykorzystane zostało łącze nazwane.

Listing 6: Przykład tworzenie i otwierania łącza nazwanego

```
#include <fcntl.h>

main() {
    int pdesk;

    if (mkfifo("/tmp/fifo", 0600) == -1){
       perror("Tworzenie kolejki FIFO");
       exit(1);
    }
```

```
    switch(fork()){
12      case -1: // blad w tworzeniu procesu
           perror("Tworzenie procesu");
           exit(1);
15      case 0:
           pdesk = open("/tmp/fifo", O_WRONLY);
           if (pdesk == -1){
18            perror("Otwarcie potoku do zapisu");
              exit(1);
           }
21         if (write(pdesk, "Hallo!", 7) == -1){
              perror("Zapis do potoku");
              exit(1);
24         }
           exit(0);
        default: {
27         char buf[10];

           pdesk = open("/tmp/fifo", O_RDONLY);
30         if (pdesk == -1){
              perror("Otwarcie potoku do odczytu");
              exit(1);
33         }
           if (read(pdesk, buf, 10) == -1){
              perror("Odczyt z potoku");
36            exit(1);
           }
           printf("Odczytano z potoku: %s\n", buf);
39      }
    }
}
```

Opis programu: Łącze nazwane (kolejka FIFO) tworzona jest w wyniku wykonania funkcji `mkfifo` w linii 6. Następnie tworzony jest proces potomny (linia 11) i łącze otwierane jest przez oba procesy (potomny i macierzysty) w sposób komplementarny (odpowiednio linia 16 i linia 29). W dalszej części przetwarzanie przebiega tak, jak w przykładzie na listingu 1.

Listing 7 jest programową realizacją potoku `ls | tr a-z A-Z`, w której wykorzystane zostało łącze nazwane podobnie, jak łącze nienazwane w przykładzie na listingu 4.

Listing 7: Programowa realizacja potoku `ls | tr a-z A-Z` na łączu nazwanym

```
#include <stdio.h>
#include <fcntl.h>
3
main(int argc, char* argv[]) {
    int pdesk;
6
    if (mkfifo("/tmp/fifo", 0600) == -1){
       perror("Tworzenie kolejki FIFO");
9      exit(1);
    }

12  switch(fork()){
       case -1: // blad w tworzeniu procesu
          perror("Tworzenie procesu");
15        exit(1);
       case 0: // proces potomny
```

```
            close(1);
18          pdesk = open("/tmp/fifo", O_WRONLY);
            if (pdesk == -1){
               perror("Otwarcie potoku do zapisu");
21             exit(1);
            }
            else if (pdesk != 1){
24             fprintf(stderr, "Niewlasciwy deskryptor do zapisu\n");
               exit(1);
            }
27          execvp("ls", argv);
            perror("Uruchomienie programu ls");
            exit(1);
30       default: { // proces macierzysty
            close(0);
            pdesk = open("/tmp/fifo", O_RDONLY);
33          if (pdesk == -1){
               perror("Otwarcie potoku do odczytu");
               exit(1);
36          }
            else if (pdesk != 0){
               fprintf(stderr, "Niewlasciwy deskryptor do odczytu\n");
39             exit(1);
            }
            execlp("tr", "tr", "a-z", "A-Z", 0);
42          perror("Uruchomienie programu tr");
            exit(1);
         }
45    }
  }
```

Opis programu: W linii 7 tworzona jest kolejka FIFO o nazwie `fifo` w katalogu `/tmp` z prawem do zapisu i odczytu dla właściciela. Kolejka ta otwierana jest przez proces potomny i macierzysty w trybie odpowiednio do zapisu i do odczytu (linia 18 i linia 33). Następnie sprawdzana jest poprawność wykonania operacji otwarcia (linie 19 i 34) oraz poprawność przydzielonych deskryptorów (linie 23 i 38. Sprawdzanie poprawności deskryptorów polega na upewnieniu się, że deskryptor łącza do zapisu ma wartość 1 (łącze jest standardowym wyjściem procesu potomnego), a deskryptor łącza do odczytu ma wartość 0 (łącze jest standardowym wejściem procesu macierzystego). Później następuje uruchomienie odpowiednio programów `ls` i `tr` podobnie, jak w przykładzie na listingu 4.

4.3 Przykłady błędów w synchronizacji procesów korzystających z łączy

Operacje zapisu i odczytu na łączach realizowane są w taki sposób, że procesy podlegają synchronizacji zgodnie ze modelem producent-konsument. Nieodpowiednie użycie dodatkowych mechanizmów synchronizacji może spowodować konflikt z synchronizacją na łączu i w konsekwencji prowadzić do stanów niepożądanych typu zakleszczenie (ang. deadlock).

Listing 8 przedstawia przykład programu, w którym może nastąpić zakleszczenie, gdy pojemność łącza okaże się zbyt mała dla pomieszczenia całości danych przekazywanych przez polecenie `ls`.

Listing 8: Możliwość zakleszczenia w operacji na łączu nienazwanym

```
1 #define MAX 512

  main(int argc, char* argv[]) {
4    int pdesk[2];
```

```
    if (pipe(pdesk) == -1){
7       perror("Tworzenie potoku");
        exit(1);
    }
10
    if (fork() == 0){ // proces potomny
        dup2(pdesk[1], 1);
13      execvp("ls", argv);
        perror("Uruchomienie programu ls");
        exit(1);
16  }
    else { // proces macierzysty
        char buf[MAX];
19      int lb, i;

        close(pdesk[1]);
22      wait(0);
        while ((lb=read(pdesk[0], buf, MAX)) > 0){
            for(i=0; i<lb; i++)
25              buf[i] = toupper(buf[i]);
            write(1, buf, lb);
        }
28  }
}
```

Opis programu: Podobnie jak w przykładzie na listingu 3 proces potomny przekazuje dane (wynik wykonania programu `ls`) do potoku (linie 12–15), a proces macierzysty przejmuje i przetwarza te dane w pętli w liniach 23–27. Przed przejściem do wykonania pętli proces macierzysty oczekuje na zakończenie potomka (linia 22). Jeśli dane generowane przez program `ls` w procesie potomnym nie zmieszczą się w potoku, proces ten zostanie zablokowany gdzieś w funkcji `write` w programie `ls`. Proces potomny nie będzie więc zakończony i tym samym proces macierzysty nie wyjdzie z funkcji `wait`. Odblokowanie potomka może nastąpić w wyniku zwolnienia miejsca w potoku przez odczyt znajdujących się w nim danych. Dane te powinny zostać odczytane przez proces macierzysty w wyniku wykonania funkcji `read` (linia 23), ale proces macierzysty nie przejdzie do linii 23 przed zakończeniem potomka. Proces macierzysty blokuje zatem potomka, nie zwalniając miejsca w potoku, a proces potomny blokuje przodka w funkcji `wait`, nie kończąc się. Wystąpi zatem zakleszczenie. Zakleszczenie nie wystąpi w opisywanym programie, jeśli wszystkie dane, generowane przez program `ls`, zmieszczą się w całości w potoku. Wówczas proces potomny będzie mógł się zakończyć po umieszczeniu danych w potoku, w następstwie czego proces macierzysty będzie mógł wyjść z funkcji `wait` i przystąpić do przetwarzania danych z potoku.

Przykład na listingu 9 pokazuje zakleszczenie w wyniku nieprawidłowości w synchronizacji przy otwieraniu łącza nazwanego.

Listing 9: Możliwość zakleszczenia przy otwieraniu łącza nazwanego

```
#include <fcntl.h>
#define MAX 512
3
main(int argc, char* argv[]) {
    int pdesk;
6
    if (mkfifo("/tmp/fifo", 0600) == -1){
        perror("Tworzenie kolejki FIFO");
9       exit(1);
    }
```

```
12    if (fork() == 0){ // proces potomny
         close(1);
         open("/tmp/fifo", O_WRONLY);
15       execvp("ls", argv);
         perror("Uruchomienie programu ls");
         exit(1);
18    }
      else { // proces macierzysty
         char buf[MAX];
21       int lb, i;

         wait(0);
24       pdesk = open("/tmp/fifo", O_RDONLY);
         while ((lb=read(pdesk, buf, MAX)) > 0){
            for(i=0; i<lb; i++)
27             buf[i] = toupper(buf[i]);
            write(1, buf, lb);
         }
30    }
  }
```

Opis programu: Proces potomny w linii 13 próbuje otworzyć kolejkę FIFO do zapisu. Zostanie on zatem zablokowany do momentu, aż inny proces wywoła funkcję `open` w celu otwarcia kolejki do odczytu. Jeśli jedynym takim procesem jest proces macierzysty (linia 23), to przejdzie on do funkcji `open` dopiero po zakończeniu procesu potomnego, gdyż wcześniej zostanie zablokowany w funkcji `wait`. Proces potomny nie zakończy się, gdyż będzie zablokowany w funkcji `open`, więc będzie blokował proces macierzysty w funkcji `wait`. Proces macierzysty nie umożliwi natomiast potomkowi wyjścia z `open`, gdyż nie może przejść do linii 23. Nastąpi zatem zakleszczenie.

4.4 Zadania

4.1. Jaki będzie wynik wykonania programu na listingu 10?

Listing 10:

```
1  main() {
       int pdesk[2];
       char buf[20];
4
       pipe(pdesk);

7      if (fork() == 0){   // proces potomny
          read(pdesk[0], buf, 20);
          printf("Odczytano z potoku: %s\n", buf);
10        write(pdesk[1], "Hallo od potomka!", 18);
          exit(0);
       }
13     else {              // proces macierzysty
          read(pdesk[0], buf, 20);
          printf("Odczytano z potoku: %s\n", buf);
16        write(pdesk[1], "Hallo od przodka!", 18);
       }
   }
```

4.2. Zrealizować na łączach nienazwanych oraz nazwanych następujące potoki:

a) `finger | cut -d' ' -f1`

b) `ps -ef | grep ^root`

c) `cat /etc/passwd | cut -f5 -d:`

d) `ls -l | grep ^d | more`

e) `ps -ef | tr -s ' ' | cut -f2 -d' '`

f) `finger | tr -s ' '| cut -f2,3 -d' '`

g) `who | cut -f1 -d' ' | sort | uniq -c`

h) `history | cut -c7- | tail -30 | sort | uniq`

i) `ps -ef | tr -s \ : | cut -f1 -d: | sort | uniq -c | sort -n`

4.3. Napisać program do zabijania wszystkich procesów użytkownika, którego nazwa podana jako argument linii poleceń.

4.4. Napisać program do obliczania łącznej liczby znaków we wszystkich nazwach plików w katalogu podanym jako argument linii poleceń.

4.5 Pytania kontrolne

1. Ile deskryptorów tworzy funkcja systemowa `pipe`?
2. Czym się różni łącze nienazwane od nazwanego? Na czym polega różnica w sposobie użycia jednego i drugiego rodzaju łącza?
3. Jakie ograniczenia na możliwość komunikacji pomiędzy procesami wprowadza potok (łącze nienazwane)?
4. W czym są podobne, a czym się różnią od siebie funkcje `creat` i `mkfifo`?
5. Co się stanie przy próbie odczytu danych z pustego łącza?
6. Czym się różni sposób dostępu do plików zwykłych od dostępu do łączy komunikacyjnych?
7. Jaka operacja dostępu do pliku (zwykłego lub specjalnego) i w jaki sposób informuje o osiągnięciu końca pliku?
8. Jakie funkcje systemowe zwracają deskryptory plików?