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 }
```