

## 4 Potoki

### 4.1 Wprowadzenie

**Potok** (ang. *pipe*) można uznać za plik specjalnego typu, który służy do przechowywania ograniczonej ilości danych i do którego dostęp może się odbywać jedynie w trybie FIFO (ang. *first-in-first-out* – element umieszczony w buforze pierwszy, również pierwszy opuści bufor). *Maksymalna liczba bajtów*, jaką można zapisać w *potoku*, jest określona stałą `PIPE_BUF`, której definicja znajduje się w pliku nagłówkowym `<limits.h>` lub `<sys/param.h>`. *Potoki* zapewniają prosty *synchroniczny* sposób *wymiany danych* między procesami. Dane *zapisywane* są na *jedym końcu potoku*, a *odczytywane* na *drugim jego końcu*, przy czym *odczytane* dane są z *potoku usuwane*.

System zapewnia *synchronizację* między procesem zapisującym i odczytującym. Domyślnie, jeśli ten pierwszy spróbuje zapisać dane do *pełnego potoku*, to zostanie przez system automatycznie *zablokowany* do czasu, gdy *potok* będzie w stanie je odebrać. Podobnie proces odczytujący, który podejmie próbę pobrania danych z *pustego potoku*, zostanie *zablokowany* do czasu, kiedy pojawią się jakieś dane. Do *zablokowania* dojdzie również wtedy, gdy *potok* zostanie *otwarty* przez jeden proces *do odczytu*, ale *nie* zostanie *otwarty* przez inny proces *do zapisu*.

Dane można *zapisywać* do *potoku* za pomocą funkcji `write`, a *odczytywać* z niego za pomocą funkcji `read` (niebuforowane funkcje wejścia/wyjścia).

Pliki włączane	<unistd.h>		
Prototyp	<code>ssize_t write(int filedes, const void *buf, size_t nbyte);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	Liczba zapisanych bajtów	−1	Tak

Funkcja `write` podejmuje próbę zapisania `nbyte` bajtów danych wskazywanych przez parametr `buf` do pliku określonego deskryptorem `filedes`. Jeśli funkcja `write` zakończy się sukcesem, to zwróci liczbę rzeczywiście zapisanych bajtów.

Pliki włączane	<unistd.h>		
Prototyp	<code>ssize_t read(int filedes, void *buf, size_t nbyte);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	Liczba odczytanych bajtów	−1	Tak

Funkcja `read` odczytuje `nbyte` bajtów danych z pliku o deskrytorze `filedes` i umieszcza je w miejscu pamięci określonym wskaźnikiem `buf`. Pozytywnie zakończona, zwraca liczbę rzeczywiście odczytanych bajtów. W przypadku dojścia do końca pliku zwracana jest wartość 0.

⇒ **Co to jest deskryptor pliku?**

W systemie UNIX z każdym otwartym przez proces plikiem związana jest pewna nieujemna liczba całkowita zwana *deskryptorem pliku*. Deskryptory o numerach 0, 1 i 2

związane są odpowiednio ze *standardowymi strumieniami*: *wejścia, wyjścia i wyjścia błędów* (mają one swoje symboliczne nazwy odpowiednio: `STDIN_FILENO`, `STDOUT_FILENO` i `STDERR_FILENO`). Deskryptory służą do odwoływania się do plików przez niektóre funkcje systemowe. Deskryptory plików procesu macierzystego są dziedziczone przez procesy potomne. Liczba deskryptorów, które mogą być dostępne procesowi jest ograniczona. W starszych wersjach Uniksa nie mogła ona przekroczyć 20, w nowszych wersjach liczba ta może być znacznie większa (można ją sprawdzić przy pomocy komendy `ulimit -n` powłoki `bash`). Deskryptor pliku można utworzyć np. przy pomocy funkcji systemowej `open`.

Pliki włączane	<sys/types.h>, <sys/stat.h>, <fcntl.h>		
Prototyp	<code>int open(const char *path, int flags, mode_t mode);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	Deskryptor pliku	-1	Tak

Pomyślnie wykonana funkcja `open` otwiera plik i zwraca jego *deskryptor*.

- Parametry:

`path`    ścieżkowa nazwa pliku,  
`flags`    opcje,  
`mode`    prawa dostępu do pliku (stosuje się zapis w systemie ósemkowym),  
           np. `rw-r--r--`    → w systemie binarnym: `110100100`  
                               → w systemie ósemkowym: `0644`

- Opcje `flags` (ważniejsze):

`O_RDONLY`    otwórz plik do czytania,  
`O_WRONLY`    otwórz plik do pisania,  
`O_RDWR`      otwórz plik do czytania i pisania,  
`O_CREAT`      jeśli plik nie istnieje, to stwórz go,  
`O_EXCL`      przy równocześnie ustawionej fladze `O_CREAT` przekaż błąd, jeśli plik już istnieje,  
`O_TRUNC`      jeśli plik istnieje, zmniejsz jego długość do zera (obetnij go),  
`O_APPEND`    otwórz plik w trybie dopisywania na jego końcu.

Opcje można łączyć przy pomocy sumy bitowej, np. `O_WRONLY | O_CREAT | O_TRUNC`.

Deskryptor przypisany do pliku można zwolnić używając funkcji systemowej `close`.

Pliki włączane	<unistd.h>		
Prototyp	<code>int close(int fildes);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	0	-1	Tak

Pomyślnie zakończona funkcja `close` zamyka plik, tzn. zwalnia używany przez niego deskryptor. Dzięki temu deskryptor ten może być użyty ponownie jeszcze przed zakończeniem wykonywania procesu. W chwili zakończenia wykonywania procesu, wszystkie używane przez ten proces pliki są automatycznie zamykane. Do dobrego stylu programowania należy jednak jawne zamykanie plików, które nie są już potrzebne (jak wiadomo,

porządek w programie jest nie tylko kwestią estetyki, ale często pozwala uniknąć różnych problemów, szczególnie przy dalszym rozwoju programu). Funkcję `close` można stosować również do deskryptorów o numerach 0, 1 i 2, tzn. *standardowych strumieni WE/WY*.

Plik można usunąć przy pomocy funkcji systemowej `unlink`.

Pliki włączane	<unistd.h>		
Prototyp	<code>int unlink(const char *path);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	0	-1	Tak

Funkcja `unlink`, pomyślnie zakończona, usuwa dowiązanie do pliku o nazwie ścieżkowej `path` z katalogu oraz zmniejsza o 1 licznik dowiązań do tego pliku przechowywany w i-węźle (*ang.* i-node) – jeśli licznik ten przyjmie wartość 0, to plik zostanie usunięty.

## 4.2 Potoki nienazwane

**Potoki nienazwane** mogą łączyć tylko procesy pokrewne, np. macierzysty i potomny, dwóch „braci”, „dziadka” i „wnuka”, itd. Do ich tworzenia służy funkcja systemowa `pipe`.

Pliki włączane	<unistd.h>		
Prototyp	<code>int pipe(int filedes[2]);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	0	-1	Tak

W przypadku poprawnego wykonania, funkcja `pipe` zwraca dwa deskryptory plików: `filedes[0]` i `filedes[1]`, które odnoszą się do dwóch strumieni danych. W obecnych wersjach systemu UNIX, funkcja `pipe` nierzadko występuje w dwóch odmianach. Jedna z nich służy do tworzenia *potoków jednokierunkowych* (półdupleks), a druga do tworzenia *potoków dwukierunkowych* (pełny dupleks). To, która z nich jest wywoływana domyślnie, zależy od ustawień systemowych – najlepiej sprawdzić to w `man pipe`. W *pełnym duplek-sie* plik `filedes[0]` służy do zapisu danych, a `filedes[1]` do ich odczytu i odwrotnie. Natomiast w konfiguracji *półdupleksowej* plik `filedes[1]` jest *zawsze* używany do *zapisu*, a `filedes[0]` *zawsze* do *odczytu* – próba użycia ich na odwrót kończy się błędem. Linux stosuje rozwiązanie półdupleksowe. Od wersji jądra o numerze 2.6.27 w Linuksie dostępna jest również specyficzna dla tego systemu funkcja `pipe2` – szczegóły można znaleźć w `man 2 pipe`.

→ UWAGA: Proces powinien zamykać (najlepiej na samym początku) koniec potoku, którego nie będzie używał (tzn. zwalniać odpowiedni deskryptor funkcją `close`) – nie jest to konieczne, ale może uchronić przed ewentualnymi błędami!

#### ĆWICZENIE 4: PRODUCENT–KONSUMENT: POTOKI NIENAZWANE

Przy pomocy **potoków nienazwanych** systemu UNIX zaimplementować problem „**Producenta i konsumenta**”. Dla zademonstrowania, że nie doszło do utraty ani zwielokrotnienia *towaru*, niech **Producent** pobiera „surowiec” (np. porcje bajtów) z *pliku tekstowego* i wstawia go jako *towar* do **potoku**, a **Konsument** niech umieszcza pobrany z **potoku** *towar* w *innym pliku tekstowym* (porcje danych **Producenta** i **Konsumenta** nie muszą być równe). Po zakończeniu działania programów (wyczerpaniu zasobów „surowca”) *oba pliki tekstowe* powinny być *identyczne* (można to sprawdzić poleceniem `diff -s`, które najlepiej umieścić w pliku `Makefile` po poleceniu uruchomienia programu). Oba procesy niech drukują odpowiednie komunikaty na ekranie, w tym *towar*, który przesyłają. Do zasymulowania różnych szybkości działania programów użyć funkcji `sleep`, np. z losowym czasem usypiania. Do czytania/pisania danych z/do pliku tekstowego, jak również wypisywania ich na ekran użyć funkcji `read` i `write`.

### 4.3 Potoki nazwane (potoki FIFO)

Drugi rodzaj potoków występujących w systemie UNIX stanowią **potoki nazwane** (określane także mianem **potoków FIFO**). Różnią się one od potoków nienazwanych głównie tym, że towarzyszą im wpisy w wykazach plików odpowiednich katalogów. Zatem w programach mogą być traktowane jak zwykłe pliki. Dzięki temu *potoki FIFO* mogą łączyć niezależne procesy, a nie tylko pokrewne jak w przypadku potoków nienazwanych. Dodatkowo *potoki FIFO* można *tworzyć* nie tylko z poziomu programu, ale również z *poziomu powłoki* (z linii poleceń). Dzięki temu *potoków FIFO* można łatwo używać w programach pisanych nie tylko w językach C/C++, ale w dowolnych językach, które posiadają operacje czytania/pisania w plikach.

Do tworzenia *potoków FIFO* z linii komend służy polecenie `mkfifo`<sup>3</sup>. Składnia tego polecenia jest następująca:

```
mkfifo [-m mode] file_name
```

gdzie *file\_name* jest nazwą *pliku specjalnego FIFO*, który będzie używany jako *potok FIFO*, a (opcjonalny) *mode* oznacza prawa dostępu (jak dla zwykłego pliku, np. 644). *Plik specjalny FIFO* w wykazie zawartości katalogu (komenda `ls -l`) jest oznaczony literą *p* (od *pipe*) w pierwszej kolumnie atrybutów pliku.

Z poziomu programu *potoki FIFO* można tworzyć przy pomocy funkcji bibliotecznej `mkfifo`<sup>4</sup>.

Pliki włączane	<sys/types.h>, <sys/stat.h>		
Prototyp	int mkfifo(const char *path, mode_t mode);		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	0	-1	Tak

Funkcja `mkfifo` tworzy *potok FIFO* o nazwie ścieżkowej *path* i prawach dostępu określonych parametrem *mode* – w systemie ósemkowym, np. 0644.

#### ĆWICZENIE 5: PRODUCENT–KONSUMENT: POTOKI NAZWANE

Przy pomocy **potoków nazwanych** systemu UNIX zaimplementować problem „**Producenta i Konsumenta**” z ćwiczenia 4.

- Utworzyć **potok FIFO** z poziomu programu, a następnie uruchomić procesy **Producenta** i **Konsumenta** w tym samym programie (w procesie macierzystym i potomnym lub w dwóch potomnych). Potok usuwać w funkcji zarejestrowanej przez `atexit`.
- Sprawdzić, że **potoki FIFO** działają dla niezależnych procesów przez utworzenie **potoku FIFO** z linii komend oraz uruchomienie procesów **Producenta** i **Konsumenta** niezależnie z poziomu powłoki, np. w różnych terminalach – można użyć polecenia:  

```
xterm -hold -title nazwa_okna -e nazwa_programu &
```

dla każdego z tych procesów w pliku `Makefile` (jak też tworzenie/usuwanie potoku).

<sup>3</sup>Istnieje również bardziej ogólne polecenie o nazwie `mknod` do tworzenia tzw. plików specjalnych, w tym m.in. *potoków FIFO* – szczegóły można znaleźć w `man mknod`.

<sup>4</sup>Istnieje także funkcja systemowa `mknod` – odpowiednik polecenia `mknod` (patrz: `man 2 mknod`).