Systemy Operacyjne 2018/2019

Potoki - materiały pomocnicze

Potoki

Potoki stanowią jeden z podstawowych mechanizmów komunikacji międzyprocesowej w systemie Linux. Potoki umożliwiają wymianę danych pomiędzy procesami w sposób przypominający wykorzystanie pliku (np. dostęp do nich odbywa się poprzez deskryptory plików), natomiast unikają narzutu fizycznych operacji dyskowych, bowiem zapis i odczyt odbywają się do/z pamięci operacyjnej.

Istnieją dwa rodzaje potoków: potoki nienazwane, oraz potoki nazwane, często nazywane też kolejkami lub FIFO. Są one podobne pod względem interfejsu służącego do ich użytkowania, natomiast spełniają nieco różne role i posiadają inne ograniczenia.

Potoki nienazwane

Potoki nienazwane są najstarszym mechanizmem komunikacji międzyprocesowej (IPC) w systemach unixowych. Zakres oferowanej przez nie funkcjonalności jest mocno ograniczony, jednak często wystarczający, co wraz ze stosunkowo prostym interfejsem sprawia, że pozostają często używane. Ze względu na opisany niżej sposób ich tworzenia, potoki nienazwane umożliwiają komunikację jedynie pomiędzy procesami posiadającymi wspólnego przodka.

Ponadto, potoki nienazwane umożliwiają komunikację tylko w jedną stronę (tzw. half-duplex) - intuicyjnie, potok ma dwa końce odpowiadające komunikującym się procesom i ustalony kierunek przepływu danych, nie można przesyłać danych "pod prąd". Standard Single UNIX Specification dopuszcza potoki dwukierunkowe (full duplex) jako rozszerzenie i niektóre systemy unixopodobne taką funkcjonalność oferują, natomiast nie jest to przenośne (np. Linux tego nie robi). Jeśli potrzebujemy przesyłać dane w obie strony, najlepiej użyć dwóch różnych potoków (albo innego mechanizmu IPC).

Tworzenie potoku nienazwanego

Do utworzenia potoku nienazwanego służy funkcja `pipe` z `<unistd.h>` o sygnaturze

```
int pipe(int fd[2]);
```

Jako argument przyjmuje ona tablicę dwóch liczb całkowitych, do których po stworzeniu potoku zapisywane są deskryptory reprezentujące "wlot", czyli końcówkę do zapisu - `fd[1]`, oraz "wylot", czyli końcówkę do odczytu - `fd[0]`. Indeksy te są spójne z numeracją wejścia i wyjścia standardowego. Wartość zwracana standardowo informuje o sukcesie (0) lub porażce (-1). Wywołanie może zakończyć się porażką, jeśli przekażemy nieprawidłowy adres tablicy, lub proces przekroczy limit ilości otwartych deskryptorów.

Ponieważ dwa procesy mogą się komunikować przez łącze tylko w jedną stronę, więc żaden z nich nie wykorzysta obydwu deskryptorów. Każdy z procesów powinien zamknąć nieużywany deskryptor łącza za pomocą funkcji **close**().

Korzystanie z potoku

Do przesyłania danych przez potok wykorzystywane są znane już funkcje `read` i `write`, do których jako deskryptory podać należy deskryptor odpowiedniego końca potoku. Domyślnie operacje odczytu i zapisu są blokujące - `read` czeka, aż w buforze potoku znajdą się jakieś dane (niekoniecznie tyle, ile zażądaliśmy), zaś `write` czeka, aż zapisane zostaną wszystkie przekazane do niego dane.

Operacje read i write na deskryptorach reprezentujących potoki (zarówno nienazwane, jak i nazwane) mają pewne specjalne zachowania:

- Operacja `read` na potoku, którego końcówka do zapisu nie ma żadnego otwartego deskryptora zwraca 0 (EOF) po przeczytaniu wszystkich danych
- Operacja `write` na potoku, którego końcówka do odczytu została zamknięta powoduje wysłanie do procesu piszącego sygnału `SIGPIPE` i zwrócenie z `write` wartości -1 plus ustawnienie `errno` na `EPIPE`. Innymi słowy, nie można pisać do potoku, z którego nikt nie czyta.

Czytanie danych z łącza przebiega zgodnie z poniższymi regułami:

- jeżeli proces chce przeczytać mniejsza ilość danych niż jest w łączu, to proces przeczyta tyle danych ile chciał
- jeżeli proces chce przeczytać większa ilość danych niż jest w łączu, to proces przeczyta tyle danych ile jest w łączu
- jeżeli łącze jest puste, to jeśli została ustawiona flaga O_NONBLOCK, zostanie zwrócony błąd EAGAIN, w przeciwnym przypadku proces zasypia dopóki łącze nie będzie puste i będzie można przeczytać dane. Jeśli nie ma pisarzy to w obydwu przypadkach zwracane jest 0

Pisanie danych do łącza przebiega zgodnie z poniższymi regułami:

- jeżeli proces chce zapisać mniejszą liczbę danych niż wynosi ilość wolnego miejsca, to zostanie zapisane do łącza tyle bajtów ile proces chciał
- jeżeli proces chce zapisać do łącza większą liczbę bajtów niż wynosi ilość wolnego miejsca, to jeśli zostala użyta flaga O_NONBLOCK, zostanie zwrócona liczba bajtów, ktorą udało się zapisać (jeśli nie udało się nic zapisać, to będzie zwrócony błąd EAGAIN), w przeciwnym przypadku proces zostanie uśpiony, aż do momentu kiedy zwolni się miejsce w łączu
- jeżeli nie ma czytelników funkcja zwróci błąd EPIPE, a do procesu zostanie wysłany sygnał SIGPIPE (standardowa obsługa sygnału SIGPIPE jest zakończenie procesu)
- Jeżeli proces chce zapisać większą liczbę bajtów niż wynosi ilość wolnego miejsca i nie jest ustawiona flaga O_NONBLOCK, to operacja pisania może nie być niepodzielna (po części danych od jednego procesu może zostać umieszczona część danych od drugiego procesu). W pozostałych przypadkach operacja pisania jest podzielna.

Użycie potoku pomiędzy procesami

Powyższe informacje pozwalają nam stworzyć potok i przesyłać za jego pomocą dane w obrębie procesu, który go stworzył. By wykorzystać taki potok do komunikacji międzyprocesowej, skorzystamy z faktu, że proces potomny otrzymuje zduplikowane deskryptory plików procesu macierzystego. Stąd, jeśli w procesie A stworzymy potok poprzez wywołanie funkcji `pipe`, a następnie stworzymy proces potomny B przy użyciu funkcji `fork`, proces B będzie posiadał otwarte deskryptory obydwu końców potoku, tak samo jak proces A.

Komunikacja rodzic - dziecko

W najprostszym wariancie mechanizm ten można wykorzystać do przesyłania danych pomiędzy rodzicem a dzieckiem. Załóżmy dla przykładu, że chcemy przesyłać dane z rodzica do dziecka. Możemy posłużyć się następującym kodem:

```
int fd[2];
pipe(fd);
pid_t pid = fork();
if (pid == 0) { // dziecko
    close(fd[1]);
    // read(fd[0], ...) - odczyt danych z potoku
} else { // rodzic
    close(fd[0]);
    // write(fd[1], ...) - zapis danych do potoku
}
```

Ze względu na zachowanie funkcji read i write dla potoków opisane wyżej najlepiej po wywołaniu funkcji `fork` zamknąć w każdym procesie nieużywaną końcówkę potoku (deskryptory są *duplikowane*, nie współdzielone, więc są niezależne dla rodzica i dziecka).

Zastosowanie łącza nienazwanego do jednokierunkowej komunikacji między procesami:

Jeden z procesów zamyka łącze do czytania a drugi do pisania. Uzyskuje się wtedy jednokierunkowe połączenie między dwoma procesami.

Zastosowanie łączy nienazwanych do dwukierunkowej komunikacji między procesami:

Dwukierunkowa komunikacja między procesami wymaga użycia dwóch łączy komunikacyjnych. Jeden z procesów pisze do pierwszego łącza i czyta z drugiego, a drugi proces postępuje odwrotnie. Obydwa procesy zamykają nieużywane deskryptory.

Komunikacja dziecko - dziecko

Analogicznie możemy wykorzystać potoki do komunikacji pomiędzy wieloma procesami potomnymi:

```
int fd[2];
pipe(fd);
if (fork() == 0) { // dziecko 1 - pisarz
    close(fd[0]);
    // ...
} else if (fork() == 0) { // dziecko 2 - czytelnik
    close(fd[1]);
    // ...
}
```

Przekierowanie wejścia i wyjścia standardowego

Powyższe sposoby są wystarczające, jeśli chcemy komunikować się z procesami, które wykonują kod programu, w którym wywołujemy funkcję fork. Czasem chcemy jednak wywołać program zewnętrzny poprzez `fork` + `exec` i np. przekazać jakieś dane na jego wejście standardowe, lub stworzyć pipeline przetwarzający dane poprzez "przepuszczenie" danych przez kilka programów. Aby tego dokonać, możemy ustawić w stworzonych procesach wejście/wyjście standardowe na odpowiednie deskryptory potoku używając funkcji `dup2`:

```
int dup2(int oldfd, int newfd);
```

Jej działanie polega na skopiowaniu deskryptora `oldfd` na miejsce deskryptora o numerze `newfd` i w razie potrzeby uprzednim zamknięciu `newfd` (chyba, że `oldfd` i `newfd` są równe, wtedy wywołanie nie robi nic). Podmiana wejścia/wyjścia standardowego sprowadza się do skopiowania deskryptora potoku na miejsce `STDIN_FILENO` / `STDOUT_FILENO`. Przykładowy kod wywołania `grep`-a i podmiany jego wejścia standardowego na potok:

```
int fd[2];
pipe(fd);
pid_t pid = fork();
if (pid == 0) { // dziecko
    close(fd[1]);
    dup2(fd[0],STDIN_FILENO);
    execlp("grep", "grep","Ala", NULL);
} else { // rodzic
    close(fd[0]);
    // write(fd[1], ...) - przesłanie danych do grep-a
}
```

Analogicznie możemy łączyć wywołania programów w ciągi, jak robi to np. shell.

Uproszczenie - popen

Powyższy przypadek użycia pojawia się na tyle często, że biblioteka standardowa udostępnia funkcje pomocnicze do jego obsługi w `<stdio.h>`:

```
FILE* popen(const char* command, const char* type);
int pclose(FILE* stream);
```

Funkcja `popen` tworzy potok, nowy proces, ustawia jego wejście lub wyjście standardowe na stosowną końcówkę potoku (zależnie od wartości argumentu `type` - `"r"` oznacza, że chcemy odczytać wyjście procesu, `"w"` że chcemy pisać na jego wejście) oraz uruchamia w procesie potomnym shell (`/bin/sh`) podając mu wartość `command` jako polecenie do zinterpretowania. Jeśli operacja ta się powiedzie, `popen` zwraca obiekt `FILE*`, którego można używać z funkcjami wejścia/wyjścia biblioteki standardowej C.

Funkcja pclose oczekuje, aż tak proces powiązany z argumentem `stream` zakończy swoje działanie. W przypadku błędu zwraca -1, w przeciwnym wypadku status zakończenia tego procesu.

Analogiczne wywołanie grep-a przy pomocy `popen`:

```
FILE* grep_input = popen("grep Ala", "w");
// fputs(..., grep_input) - przesłanie danych do grep-a
pclose(grep_input);
```

Potoki nazwane

Jednym z ograniczeń potoków nienazwanych jest to, że pozwalają na komunikację jedynie pomiędzy procesami posiadającego wspólnego przodka (który musi stworzyć potok). Potoki nazwane omijają to ograniczenie poprzez wykorzystanie systemu plików do identyfikacji potoku - potok nazwany jest reprezentowany przez pewien plik na dysku. Dzięki temu wykorzystywać potok nazwany mogą dowolne procesy, bez konieczności pokrewieństwa.

Uwaga: Należy pamiętać, że wciąż nie jest to to samo, co wykorzystanie zwykłego pliku do przekazywania danych pomiędzy procesami. Plik służy tu tylko do identyfikacji potoku nazwanego, operacje zapisu i odczytu wciąż operują tylko na buforze w pamięci. Plik reprezentujący potok nazwany nie jest plikiem regularnym (`S_IFREG`), a plikiem specjalnym typu FIFO (`S_IFIFO`).

Tworzenie potoku nazwanego

Dostępne są dwa narzędzia do utworzenia pliku reprezentującego potok nazwany - mkfifo oraz mknod. Oba są dostępne zarówno jako program który można wywołać w terminalu, jak i jako wywołanie systemowe. Narzędzie `mknod` jest bardziej ogólne - potrafi tworzyć pliki specjalne różnych typów, `mkfifo` natomiast tworzy wyłącznie potoki nazwane.

Tworzenie potoku w terminalu

Utworzyć potok o nazwie NAZWA możemy następującymi poleceniami:

```
mkfifo NAZWA
mknod NAZWA p
```

Usunąć potok nazwany możemy tak, jak każdy inny plik:

```
rm NAZWA
```

Tworzenie potoku w programie

Funkcje mkfifo i mknod wymagają dołączenia `<sys/types.h>` oraz `<sys/stat.h>`:

```
int mkfifo(const char *pathname, mode_t mode);
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Podobnie jak dla funkcji open, istnieje wariant z sufixem _at, który pozwala stworzyć potok nazwany o zadanej nazwie w folderze reprezentowanym przez deskryptor. Te warianty wymagają `<fcntl.h>`:

```
int mkfifoat(int dirfd, const char *pathname, mode_t mode);
int mknodat(int dirfd, const char *pathname, mode_t mode, dev_t dev);
```

Argument mode działa jak w przypadku funkcji open, z tą różnicą że do funkcji typu mknod w celu stworzenia potoku nazwanego przekazać powinniśmy dodatkowo z-or-owaną flagę `S_IFIFO`. Argument `dev` jest wówczas ignorowany, więc można przekazać np. 0.

Używanie potoków nazwanych

Aby móc używać potoku nazwanego, musimy otworzyć plik go reprezentujący - `open` / `fopen`. Podobnie jak w przypadku potoków nienazwanych, odczyt i zapis odbywa się przy użyciu tych samych funkcji, co w przypadku zwykłych plików - `read`/`write` lub funkcje biblioteczne. Reguły są analogiczne do tych przy potoku nazwanym - nie można pisać do potoku, którego nikt nie czyta, przeczytanie wszystkiego z potoku do którego nikt już nie pisze daje efekt taki, jak napotkanie końca pliku.

Domyślnie otwieranie potoku nazwanego w trybie do odczytu blokuje do momentu, gdy jakiś inny proces otworzy potok w trybie do zapisu. Analogicznie w drugą stronę - otwieranie w trybie do zapisu blokuje do momentu, gdy ktoś inny otworzy w trybie do odczytu.

Uwaga odnośnie wielu procesów (> 2)

W przypadku obu rodzajów potoków możliwe jest skonstruowanie sytuacji, w której wiele procesów pisze do jednego końca potoku i/lub wiele procesów czyta z drugiego końca potoku. W szczególności potok nazwany można próbować wykorzystać jako bufor do którego jedna grupa procesów wpisuje jakieś jednostki danych (np. wyniki obliczeń), a druga grupa je pojedynczo odczytuje i przetwarza.

Problemem w takiej sytuacji staje się atomiczność operacji wejścia/wyjścia. Jeśli procesy A i B próbują jednocześnie zapisywać dane,

nie ma w ogólności gwarancji, że całość danych z procesu A zostanie zapisana przed całością danych z procesu B lub odwrotnie - może wystąpić sytuacja, w której zapisany do potoku zostanie kawałek danych procesu A, potem kawałek danych procesu B itd. Podobna sytuacja może mieć miejsce z odczytami.

W przypadku operacji zapisu, POSIX gwarantuje, że żądania zapisu danych wielkości nie większej, niż wartość stałej `PIPE_BUF` (co najmniej 512) z pliku `<limits.h>` będą wykonane atomicznie. Wartości `PIPE_BUF` dla różnych systemów można zobaczyć tutaj (dla Linuxa jest to 4096). Jeśli możemy zatem zagwarantować, że wszystkie procesy piszące do potoku nazwanego zachowują to ograniczenie, wieloprocesowe zapisy są bezpieczne.

W przypadku operacji odczytu nie ma tego rodzaju gwarancji, w związku z czym najlepiej unikać wielu procesów czytających z tego samego potoku nazwanego, a w razie potrzeby zaimplementowania tego rodzaju komunikacji wykorzystać inne mechanizmy.

Ostatnia modyfikacja: piątek, 5 kwiecień 2019, 00:32

◄ Zadania - Zestaw 4

Przejdź do... Przejdź do...

Zadania - Zestaw 5 ►

SO2019
Uczestnicy
Odznaki
Kompetencje
Oceny
Główne składowe
Kolokwium 1
Kolokwium 2
Laboratorium 1
Laboratorium 2
Laboratorium 3
Laboratorium 4
Laboratorium 5
Laboratorium 6
Laboratorium 7
Laboratorium 8
Laboratorium 9
Laboratorium 10
Strona główna
Kokpit
Kalendarz
Prywatne pliki
Moje kursy
to1-2019
TMP_S_1819
PBDw-2018-2019
PSI 2018/19
Statystyka2018-19