Trzy strumienie: stdin, stdout, stderr

Poziom: podstawy **Dotyczy: wszystkie dystrybucje** **Czas: ~12 min**

Każdy proces w Linuksie startuje z trzema otwartymi kanałami: wejściem, wyjściem i osobnym kanałem błędów. Skąd się biorą, dlaczego błędy jadą osobno i co z tego wynika przy każdym potoku, jaki napiszesz.

Skróty użyte na tej stronie (5)

Skrót	Co znaczy
ANSI	American National Standards Institute: tu chodzi o kody sterujące kolorem i pozycją kursora w terminalu
CSV	w Windowsie: Cluster Shared Volume, dysk widoczny naraz dla wszystkich węzłów klastra; w danych: Comma-Separated Values, plik z wartościami oddzielonymi przecinkami
PID	Process ID: numer procesu w systemie
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa

Fiszka

ZMIENIA SYSTEM

Wyciszenie błędów, zachowanie danych

```
ls /etc /nieistnieje 2>/dev/null
```

Deskryptor 2 (stderr) leci do `/dev/null`, deskryptor 1 (stdout) zostaje bez zmian. Najczęstszy sposób pozbycia się szumu typu „Permission denied" z wyniku polecenia.

Przykład:

```
/etc:
adjtime
aliases
alternatives
...
```

ls = list: wypisz zawartość katalogu

ZMIENIA SYSTEM

Wyciszenie danych, zachowanie błędów

```
ls /etc /nieistnieje 1>/dev/null
```

Odwrotność poprzedniego — przydaje się, gdy interesuje nas wyłącznie to, co poszło nie tak. `1>` i `>` znaczą dokładnie to samo; jedynka jest tu dopisana jawnie tylko dla symetrii z `2>`.

Przykład:

```
ls: cannot access '/nieistnieje': No such file or directory
```

ls = list: wypisz zawartość katalogu

ZMIENIA SYSTEM

Cisza absolutna

```
komenda > /dev/null 2>&1
```

Oba strumienie znikają: dane trafiają do `/dev/null`, a błędy tam, gdzie w tym momencie wskazuje deskryptor 1 — czyli też do `/dev/null`. Kolejność ma tu znaczenie — więcej w sekcji *Dlaczego kolejność `2>&1` ma znaczenie* niżej. Komenda nie wypisze nic.

ZMIENIA SYSTEM

To samo krócej

```
komenda &> /dev/null
```

Bashowy skrót na powyższe — jeden operator zamiast dwóch. To nie jest POSIX: w skryptach zaczynających się od `#!/bin/sh`, gdzie `sh` bywa `dash` albo `busybox sh`, może nie zadziałać jak przekierowanie.

POSIX = Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

ZMIENIA SYSTEM

Wciągnięcie błędów do potoku

```
komenda 2>&1 | grep wzorzec
```

Bez `2>&1` `grep` widziałby wyłącznie stdout — błędy lecą obok potoku prosto na ekran i `grep` nigdy by ich nie zobaczył.

Przykład:

```
błąd: nie można otworzyć pliku konfiguracyjnego
```

TYLKO ODCZYT

Podgląd otwartych deskryptorów procesu

```
ls -l /proc/$$/fd
```

`$$` to PID bieżącej powłoki. Pokazujemy, na co realnie wskazują deskryptory 0, 1 i 2 — nieoceniona pierwsza czynność, gdy usługa „nie loguje", a nie wiadomo dlaczego.

Przykład:

```
total 0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 0 -> /dev/pts/0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 1 -> /dev/pts/0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 2 -> /dev/pts/0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 255 -> /dev/pts/0
```

PID = Process ID: numer procesu w systemie · **ls** = list: wypisz zawartość katalogu

TYLKO ODCZYT

To samo dla dowolnego procesu

```
ls -l /proc/NumerPID/fd
```

Identyczna komenda, ale dla cudzego PID-u (musimy być jego właścicielem albo mieć roota). Tak wygląda usługa, która „nie loguje": deskryptor 2 wskazuje na `/dev/null` zamiast na plik z logami.

Przykład:

```
total 0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:24 0 -> /dev/null
l-wx------ 1 adm_user adm_user 64 Sep 17 10:24 1 -> /var/log/aplikacja/access.log
l-wx------ 1 adm_user adm_user 64 Sep 17 10:24 2 -> /dev/null
```

ls = list: wypisz zawartość katalogu

TYLKO ODCZYT

Wymuszenie kolorów mimo potoku

```
ls --color=always /etc | less -R
```

`ls` domyślnie gasi kolory, gdy jego wyjście nie jest terminalem — nawet jeśli ostatecznie i tak ląduje na ekranie, jak tutaj przez `less`. `--color=always` wymusza sekwencje kolorów bez względu na to, dokąd prowadzi stdout, a `less -R` interpretuje je zamiast pokazywać jako surowy tekst. Dowód, że kolory naprawdę tam są, jest w sekcji *isatty* niżej.

ls = list: wypisz zawartość katalogu

TYLKO ODCZYT

To samo dla `grep`

```
grep --color=always wzorzec plik | less -R
```

Ten sam mechanizm co wyżej, inne narzędzie. Bez `-R` po stronie `less` dostajemy ekran pełen sekwencji `^[[01;31m` zamiast podświetlonych dopasowań.

grep = global regular expression print: szukaj wzorca w tekście

□□ TYLKO ODCZYT

Wymuszenie buforowania liniowego w potoku

```
tail -F /var/log/messages | stdbuf -oL awk '{print $5}'
```

Bez `stdbuf -oL` `awk` w potoku zbiera dane w buforze blokowym i wypisuje je kawałkami po kilka kilobajtów, a nie na bieżąco. `-oL` wymusza zrzucanie bufora wyjściowego po każdej linii, więc widzimy wynik w tym samym tempie, w jakim rosną wpisy w logu.

□□ **tail** = pokaż koniec pliku, z flagą -f śledzi plik na żywo

□□ TYLKO ODCZYT

Pochłaniacz danych: /dev/null

```
cat /dev/null
```

Odczyt z `/dev/null` daje natychmiastowy koniec pliku. `cat` kończy się od razu, bez błędu i bez śladu na ekranie.

□□ **cat** = concatenate: sklej i wypisz pliki

□□ TYLKO ODCZYT

Źródło zer: /dev/zero

```
head -c 16 /dev/zero | od -An -tx1
```

`/dev/zero` przy odczycie produkuje nieskończony strumień bajtów `0x00`. `head -c 16` ucina go po 16 bajtach — bez tego polecenie czekałoby w nieskończoność. `od -An -tx1` pokazuje te bajty jako czytelny zapis szesnastkowy zamiast wysyłać surowe zera na terminal.

Przykład:

```
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

□□ **head** = pokaż początek pliku, domyślnie dziesięć pierwszych linii

□□ ZMIENIA SYSTEM

Symulacja braku miejsca na dysku: /dev/full

```
echo cokolwiek > /dev/full
```

Zapis do `/dev/full` zawsze kończy się błędem `ENOSPC`, niezależnie od ilości danych. Służy wyłącznie do testowania, czy skrypt faktycznie sprawdza kod wyjścia przy zapisie, zamiast zakładać, że skoro `echo` się wykonało, to dane naprawdę trafiły na dysk.

Przykład:

```
-bash: echo: write error: No space left on device
```

Deskryptor	Nazwa	Domyślnie podłączony do	Do czego służy
`0`	stdin	klawiatura terminala	dane wejściowe
`1`	stdout	ekran terminala	wynik pracy programu
`2`	stderr	ekran terminala	błędy, ostrzeżenia, paski postępu

Urządzenie	Odczyt daje	Zapis powoduje
`/dev/null`	natychmiastowy koniec pliku	dane znikają, sukces
`/dev/zero`	nieskończony ciąg bajtów `0x00`	dane znikają, sukces
`/dev/full`	nieskończony ciąg bajtów `0x00`	błąd `ENOSPC` — do testów

Uważaj

- `2>&1 > plik` **nie** wysyła błędów do pliku — patrz *Częste błędy*. Poprawna kolejność to `> plik 2>&1`.
- `2>&1` to „skopiuj to, gdzie **aktualnie** wskazuje 1", a nie „skleruj 2 z 1 na zawsze".
- Program, który widzi potok zamiast terminala, zmienia zachowanie: gasi kolory i przełącza się na buforowanie blokowe.

Z życia wzięte

□□ Z ŻYCIA WZIĘTE

> Poniedziałek, 11:30. Aplikacja Oli na `app01` „nie loguje", chociaż w kodzie są wywołania logujące błędy. Tomek sprawdza `ls -l /proc/PID/fd` i widzi, że deskryptor 2 wskazuje na `/dev/null`, bo ktoś tak skonfigurował jednostkę systemd. Po zmianie na plik logu błędy zaczynają się pojawiać, a Ola w końcu widzi, co naprawdę się wywala.

Po co to istnieje

> **Pojęcie: standardowe wejście, wyjście i błędy.** Każdy proces w Uniksie dostaje od jądra trzy domyślnie otwarte kanały, zanim wykona pierwszą linię swojego kodu: standardowe wejście (dane, które program czyta), standardowe wyjście (normalny wynik jego pracy) i standardowe wyjście błędów (komunikaty diagnostyczne, ostrzeżenia, paski postępu). Program nie musi ich otwierać ani zamykać — po prostu z nich korzysta. Domyślnie wszystkie trzy prowadzą do tego samego terminala, więc dopóki nic nie przekierujesz, wyglądają jak jeden wspólny ekran.

Uruchamiasz nocny skrypt, który zbiera dane z dziesięciu serwerów i wrzuca je do pliku CSV:

```
./zbierz-metryki.sh > /var/tmp/raport.csv
```

Rano plik ma 2400 wierszy i cztery linie, które psują import do bazy:

```
ssh: connect to host srv07 port 22: Connection timed out
ssh: connect to host srv07 port 22: Connection timed out
Warning: Permanently added 'srv09' to the list of known hosts.
srv03;cpu;12
```

Gdyby komunikaty o błędach leciały tym samym kanałem co dane, nie dałoby się ich rozdzielić inaczej niż przez zgadywanie wyrażeniem regularnym. Dlatego w Uniksie od początku istnieją **dwa osobne wyjścia**: jedno na wynik pracy, drugie na komentarz do tej pracy. To rozdzielenie jest warunkiem, żeby potoki miały sens — `komenda1 | komenda2` przepycha dalej wyłącznie strumień 1, a błędy lecą obok potoku prosto na ekran.

Analogia

Wyobraź sobie **taśmę produkcyjną z dwoma wyjściami**.

Z jednego końca wjeżdża surowiec — to jest **stdin**. Maszyna coś z nim robi. Gotowe, sprawdzone produkty jadą taśmą główną do następnej maszyny w hali — to **stdout**. Ale maszyna ma jeszcze

drugi, boczny zsyp: tam ląduje wszystko, co nie jest produktem. Braki, komunikat „zabrakło kleju", karteczka „partia 47 miała wilgotny materiał". To **stderr**, i ten zsyp prowadzi do kontroli jakości, a nie do następnej maszyny.

Cała hala jest zbudowana na tym założeniu: następna maszyna w ciągu obrabia wyłącznie produkty. Gdyby na taśmę główną trafiła karteczka z komunikatem, maszyna spróbowałaby obrobić ją jak detal. Dokładnie to robi `grep`, kiedy dostanie w strumieniu danych linijkę `Permission denied` — potraktuje ją jak dane, bo nie ma jak rozpoznać, że danymi nie jest.

Analogia niesie też ograniczenie, które tłumaczy realną pułapkę. **Oba zsypy domyślnie kończą się w tym samym miejscu** — na hali, przy nogach operatora. Stojąc obok maszyny, widzisz jeden wspólny bałagan i wcale nie jest oczywiste, że to dwa niezależne kanały. Tak samo jest z terminalem: stdout i stderr są domyślnie podpięte do tego samego ekranu, więc ich odrębność ujawnia się dopiero wtedy, gdy któryś przekierujesz. Do tego momentu większość ludzi nie ma powodu podejrzewać, że w ogóle istnieją dwa.

Jak to działa

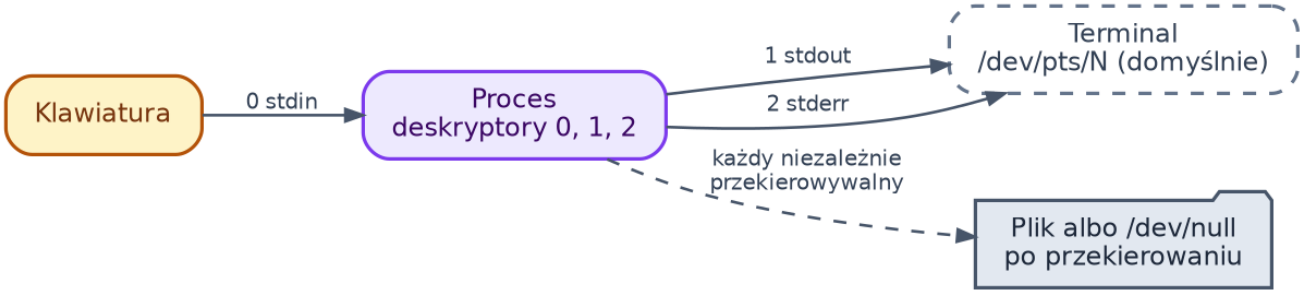
Rozwiń: jak to działa pod spodem

> **Pojęcie: deskryptor pliku.** Liczba całkowita, którą jądro przypisuje każdemu otwartemu „czemuś" w procesie — plikowi, potokowi, gniazdu sieciowemu, terminalowi. To indeks w prywatnej tablicy danego procesu, nie globalny identyfikator współdzielony z innymi programami. Deskryptory 0, 1 i 2 mają ustaloną konwencją rolę, ale każdy kolejny otwarty plik dostaje kolejny wolny numer, zwykle od 3 w górę.

Kiedy jądro tworzy nowy proces, ten dziedziczy po rodzicu tablicę otwartych deskryptorów plików. Konwencja mówi, że trzy pierwsze pozycje mają ustalone znaczenie:

- `0` — standardowe wejście (stdin)
- `1` — standardowe wyjście (stdout)
- `2` — standardowe wyjście błędów (stderr)

To jest konwencja, a nie prawo fizyki. Program może zamknąć deskryptor 1 i otworzyć nim plik — i tak właśnie działa przekierowanie. Ale cały ekosystem uniksowy zakłada tę trójkę, więc jej łamanie kończy się dziwnymi błędami.



Trzy deskryptory procesu: stdin z klawiatury, stdout i stderr domyślnie na terminal, każdy niezależnie przekierowywalny (kliknięcie otwiera pełny rozmiar)

Zobacz to na żywo. `$$` to PID bieżącej powłoki:

```
ls -l /proc/$$/fd
```

```
total 0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 0 -> /dev/pts/0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 1 -> /dev/pts/0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 2 -> /dev/pts/0
lrwx------ 1 adm_user adm_user 64 Sep 17 10:22 255 -> /dev/pts/0
```

Wszystkie trzy wskazują na `/dev/pts/0` — ten sam terminal. Stąd wrażenie, że to jeden kanał.

Gdy proces ma przekierowane strumienie, zobaczysz tu konkretne pliki zamiast `/dev/pts/0`, a uprawnienia po lewej powiedzą Ci, w jakim trybie deskryptor został otwarty: `lr-x` to odczyt, `l-wx` zapis. Ta komenda jest jednym z najbardziej niedocenianych narzędzi diagnostycznych. Jeśli usługa „nie loguje", sprawdź, dokąd naprawdę wskazuje jej deskryptor 2 — bywa, że do `/dev/null`, bo ktoś tak napisał plik jednostki systemd.

Rozdzielenie w praktyce

Klasyczna demonstracja — `ls` na dwóch ścieżkach, z których jedna nie istnieje:

```
ls /etc /nieistnieje
```

```
ls: cannot access '/nieistnieje': No such file or directory
/etc:
adjtime
```

```
aliases
alternatives
...
```

Na ekranie to wygląda jak jeden blok tekstu. Ale to są dwa niezależne strumienie. Wyrzuć błędy:

```
ls /etc /nieistnieje 2>/dev/null
```

```
/etc:
adjtime
aliases
alternatives
...
```

Teraz odwrotnie — wyrzuć dane, zostaw błędy:

```
ls /etc /nieistnieje 1>/dev/null
```

```
ls: cannot access '/nieistnieje': No such file or directory
```

`1>` i `>` znaczą dokładnie to samo; jedynka jest wartością domyślną. Piszę ją tutaj jawnie, żeby symetria z `2>` była widoczna.

Dlaczego kolejność `2>&1` ma znaczenie

Zapis `2>&1` czytaj dosłownie: „ustaw deskryptor 2 na to samo, na co **w tej chwili** wskazuje deskryptor 1". To jest kopiowanie wskaźnika, jednorazowe, w momencie przetwarzania przez powłokę. Nie jest to trwałe powiązanie.

Powłoka przetwarza przekierowania **od lewej do prawej**. Porównaj:

```
komenda > plik.txt 2>&1
```

1. `> plik.txt` — deskryptor 1 zaczyna wskazywać na `plik.txt`.
2. `2>&1` — deskryptor 2 zaczyna wskazywać tam, gdzie 1, czyli na `plik.txt`.

Efekt: oba strumienie w pliku. To jest wersja POSIX-owa i działa wszędzie.

```
komenda 2>&1 > plik.txt
```

1. `2>&1` — deskryptor 2 zaczyna wskazywać tam, gdzie 1, czyli **na terminal** (bo 1 jeszcze nie został przekierowany).
2. `> plik.txt` — deskryptor 1 zaczyna wskazywać na `plik.txt`. Deskryptor 2 zostaje tam, gdzie był: na terminalu.

Efekt: dane w pliku, błędy na ekranie. Czasami dokładnie tego chcesz — ale prawie nigdy nie o to chodzi komuś, kto to napisał.

isatty — program wie, czy patrzysz

Przy starcie wiele programów woła `isatty(1)`, czyli pyta jądro: „czy mój deskryptor 1 to terminal?". Odpowiedź zmienia ich zachowanie.

`ls` przy `--color=auto` (a to jest alias ustawiony domyślnie na RHEL, Rocky, Debianie i Ubuntu) koloruje wyjście tylko wtedy, gdy trafia ono na terminal. W potoku kolory gasną — i słusznie, bo sekwencje sterujące ANSI zepsułyby dane następnej komendzie.

Sprawdź to. Najpierw na terminalu, potem przez `cat -v`, które pokazuje znaki sterujące jako tekst:

```
ls /etc | head -2 | cat -v
ls --color=always /etc | head -2 | cat -v
```

```
ImageMagick
NetworkManager
^[[0m^[[01;34mImageMagick^[[0m
^[[01;34mNetworkManager^[[0m
```

`^[[01;34m` to sekwencja „niebieski, pogrubiony". W pierwszej wersji jej nie ma, w drugiej jest.

To samo dotyczy `grep`:

```
grep --color=always -i error /var/log/messages | less -R
```

`--color=always` wymusza kolory mimo potoku, a `less -R` przepuszcza sekwencje kolorów zamiast pokazywać je jako `^[[01;31m`. Bez `-R` dostaniesz ekran pełen śmieci.

Możesz sam sprawdzić, po której stronie jesteś:

```
[ -t 1 ] && echo "stdout to terminal" || echo "stdout to nie terminal"
```

Ten sam mechanizm stoi za tym, że `git log` odpala pager na terminalu, a w potoku nie; że `docker`, `dnf` i `apt` rysują paski postępu tylko na żywo; i że `python -c 'print(1)'` w potoku zachowuje się inaczej pod względem buforowania.

Urządzenia znakowe: `/dev/null`, `/dev/zero`, `/dev/full`

> **Pojęcie: urządzenie znakowe.** Plik w `/dev`, za którym nie stoi żaden prawdziwy nośnik danych, tylko sterownik jądra reagujący na odczyt i zapis. `/dev/null` przy odczycie natychmiast zwraca koniec pliku, a przy zapisie po cichu odrzuca wszystko, co dostał. `/dev/zero` przy odczycie generuje nieskończony ciąg bajtów zerowych. Z punktu widzenia programu to zwykły plik — `open()`, `read()`, `write()` działają tak samo — różnicę robi to, co jest po drugiej stronie.

`/dev/null` to najczęściej używane urządzenie znakowe w całym systemie. Przekierowanie `2>/dev/null` nie „usuwa" błędów w żadnym specjalnym sensie — po prostu podpina deskryptor 2 pod plik, który każdy zapis odbiera i wyrzuca, zwracając przy tym sukces. Stąd `komenda 2>/dev/null; echo $?` może pokazać `0`, nawet jeśli sama komenda wewnętrznie zgłosiła błąd — zniknął tylko tekst, kod wyjścia to osobna sprawa.

`/dev/zero` działa w drugą stronę: nie da się z niego nic „wyczytać do końca", bo sterownik zawsze ma gotowy kolejny bajt `0x00`. Używa się go do tworzenia plików o zadanym rozmiarze wypełnionych zerami (`dd if=/dev/zero of=plik bs=1M count=100`) albo do testowania przepustowości, bo generowanie danych nie kosztuje odczytu z dysku.

`/dev/full` symuluje pełny dysk przy każdym zapisie — rzadziej używane w codziennej pracy, ale nieocenione przy testowaniu, czy skrypt faktycznie sprawdza kod wyjścia zapisu, zamiast zakładać, że skoro polecenie się wykonało, dane na pewno trafiły na dysk.

Buforowanie

> **Pojęcie: buforowanie.** Odkładanie danych w pamięci zamiast natychmiastowego zapisu, żeby zamienić dużo małych, kosztownych wywołań systemowych na kilka dużych. Biblioteka standardowa C robi to automatycznie dla stdout i stdin, ale nie dla stderr — stąd błędy pojawiają się na ekranie natychmiast, a zwykłe dane czasem z opóźnieniem.

Tryb bufora zależy od tego, dokąd idzie strumień:

Cel strumienia	Tryb	Kiedy dane faktycznie wychodzą
terminal (stdout)	liniowy	po każdym znaku nowej linii
potok lub plik (stdout)	blokowy, zwykle 4 kB	po zapełnieniu bufora albo na końcu procesu
stderr	niebuforowany	natychmiast, bajt po bajcie

Stąd dwa zjawiska, które wyglądają jak awaria. Pierwsze: `tail -F log | grep error` nie pokazuje nic przez godzinę, mimo że błędy lecą — `grep` zbiera je w 4-kilobajtowym buforze. Naprawa: `grep --line-buffered` albo uniwersalnie `stdbuf -oL`. Drugie: w logu skryptu linie stderr i stdout są poprzestawiane, bo stderr wychodzi natychmiast, a stdout czeka w buforze do końca procesu. Szczegóły na osobnej stronie o `stdbuf`.

Krok po kroku

1. Wycisz tylko błędy.

```
find / -name '*.conf' 2>/dev/null | head -5
```

```
/etc/dnf/dnf.conf
/etc/fonts/fonts.conf
/etc/fstab.conf
/etc/gai.conf
/etc/host.conf
```

Bez `2>/dev/null` ekran zalewa setka linii `find: '/proc/1234/task': Permission denied`, w której gubi się wynik.

2. Wycisz wszystko.

```
systemctl is-active nginx > /dev/null 2>&1
echo $?
```

```
0
```

Tu nie interesuje Cię tekst, tylko kod wyjścia. `0` znaczy „usługa działa".

3. Wciągnij błędy do potoku.

```
./skrypt.sh 2>&1 | grep -i 'error\|fail'
```

```
2026-09-17 03:00:04 ERROR nie można połączyć z bazą danych
2026-09-17 03:00:09 FAIL   zadanie zakończone niepowodzeniem
```

Bez `2>&1` `grep` nie zobaczy błędów w ogóle — polecą obok potoku na ekran.

Częste błędy

Zła kolejność przy `2>&1`.

Piszesz skrypt crona i chcesz mieć wszystko w jednym logu:

```
0 3 * * * /opt/skrypty/backup.sh 2>&1 > /var/log/backup.log
```

Log jest, dane w nim są, ale błędów nie ma — a cron wysyła Ci codziennie mail z treścią `rsync: connection unexpectedly closed`. Powód: w momencie przetwarzania `2>&1` deskryptor 1 wskazywał jeszcze na wyjście crona (czyli na mail), więc tam poszły błędy. Dopiero potem 1 trafił do pliku.

Poprawnie:

```
0 3 * * * /opt/skrypty/backup.sh > /var/log/backup.log 2>&1
```

Reguła do zapamiętania: `2>&1` **zawsze na końcu**, po wszystkich przekierowaniach stdout.

`2>/dev/null` użyte jako sposób na pozbycie się problemu.

```
tar -czf /backup/dane.tar.gz /srv/dane 2>/dev/null
```

Backup „przechodzi bez błędów" przez pół roku. Potem okazuje się, że `tar` przez cały ten czas pisał `tar: /srv/dane/db: Cannot open: Permission denied` i połowa katalogów nigdy nie trafiła do archiwum. Wyciszenie stderr nie naprawia błędu, tylko ukrywa dowód.

Jeśli musisz wyciszyć ekran, zapisz błędy do pliku i sprawdzaj kod wyjścia:

```
tar -czf /backup/dane.tar.gz /srv/dane 2>>/var/log/backup-bledy.log || echo "BŁĄD backupu, kod $?"
```

Mylenie `2>&1` z `2>1`.

```
komenda 2>1
```

Zero błędu, wszystko wygląda w porządku — a w katalogu pojawił się plik o nazwie `1` z treścią komunikatów. Ampersand nie jest ozdobnikiem: `&1` znaczy „deskryptor numer 1", samo `1` znaczy „plik o nazwie 1".

Ta sama pułapka w drugą stronę: `komenda >&2` wysyła stdout na stderr (przydatne w funkcjach zgłaszających błąd), a `komenda > 2` tworzy plik `2`.

Założenie, że brak wyjścia znaczy brak danych.

```
tail -F /var/log/app.log | grep -i timeout
```

Czekasz dwadzieścia minut, nic się nie pojawia, więc raportujesz, że timeoutów nie ma. W rzeczywistości `grep` uzbierał 1,8 kB dopasowań i trzyma je w buforze blokowym, bo jego wyjście to potok (w tej postaci — do terminala, ale gdy dopiszesz `| tee`, już nie). Przy diagnozie na żywo zawsze:

```
tail -F /var/log/app.log | grep --line-buffered -i timeout | tee /tmp/trafienia.txt
```

Przekierowanie stdout, gdy program pisze na stderr.

```
curl -s https://bkp01.zielona.example/plik.iso > pobrane.iso
```

Jeśli `curl` zgłosi problem, `pobrane.iso` będzie pustym plikiem, a komunikat zniknie wraz z `-s`. Podobnie `wget` domyślnie cały postęp i wszystkie komunikaty wypisuje na **stderr**, więc `wget URL > log.txt` daje pusty `log.txt`. Program pisze tam, gdzie uznał za stosowne — sprawdź, zanim przekierujesz.

□□ Ćwiczenia

> Wszystko poniżej wykonujesz na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora i nic nie zostawia śladu poza plikami, które sam tworzysz.

Ćwiczenie 1 — rozdziel strumienie

```
cd /tmp
mkdir -p cwiczenie-strumienie && cd cwiczenie-strumienie
ls /etc /nieistnieje > out.txt 2> err.txt
wc -l out.txt err.txt
cat err.txt
```

Sprawdź, czy `out.txt` zawiera wyłącznie listę plików, a `err.txt` wyłącznie jedną linię błędu.

Odpowiedź

`err.txt` ma dokładnie jedną linię:

```
ls: cannot access '/nieistnieje': No such file or directory
```

`out.txt` ma tyle linii, ile jest pozycji w `/etc` plus linia nagłówka `/etc:`. Żadna z nich nie zawiera słowa `cannot`. Sprawdź to wprost:

```
grep -c cannot out.txt
```

Wynik: `0`.

Ćwiczenie 2 — udowodnij, że kolejność ma znaczenie

```
cd /tmp/cwiczenie-strumienie
ls /etc /nieistnieje > A.txt 2>&1
ls /etc /nieistnieje 2>&1 > B.txt
grep -c cannot A.txt B.txt
```

Odpowiedź

```
A.txt:1
B.txt:0
```

Przy `B.txt` komunikat błędu wylądował na terminalu — zobaczyłeś go między dwoma komendami. Deskryptor 2 został skopiowany z jedynki, zanim jedynka trafiła do pliku.

Ćwiczenie 3 — zajrzyj procesowi do deskryptorów

```
cd /tmp/cwiczenie-strumienie
sleep 120 > wyjscie.log 2> bledy.log < /dev/null &
PID=$!
ls -l /proc/$PID/fd
kill $PID
```

Odpowiedź

```
lr-x------ 1 adm_user adm_user 64 Sep 17 11:03 0 -> /dev/null
l-wx------ 1 adm_user adm_user 64 Sep 17 11:03 1 -> /tmp/cwiczenie-strumienie/wyjscie.log
l-wx------ 1 adm_user adm_user 64 Sep 17 11:03 2 -> /tmp/cwiczenie-strumienie/bledy.log
```

Dowód, że przekierowanie to nie magia powłoki, tylko konkretny stan tablicy deskryptorów procesu — ustawiony przez powłokę tuż przed `exec`.

Ćwiczenie 4 — urządzenia specjalne

```
head -c 8 /dev/zero | od -An -tx1
wc -c < /dev/null
echo test > /dev/full; echo "kod wyjścia: $?"
```

Odpowiedź

```
 00 00 00 00 00 00 00 00
0
-bash: echo: write error: No space left on device
kod wyjścia: 1
```

`/dev/zero` produkuje bajty zerowe bez końca, `/dev/null` przy odczycie daje natychmiastowy EOF, a `/dev/full` służy do testowania, czy Twój skrypt w ogóle sprawdza kody wyjścia przy zapisie. Jeśli skrypt zapisuje do `/dev/full` i twierdzi, że wszystko poszło dobrze — masz w nim błąd.

Pytania kontrolne

1. Napisałeś `komenda 2>&1 | grep blad` i widzisz dopasowania. Kolega napisał `komenda | grep blad 2>&1` i nie widzi nic. Dlaczego?

Odpowiedź

W wersji kolegi `2>&1` dotyczy `grep`, a nie `komenda`. Przekierowanie wiąże się z tą komendą, przy której stoi — a stderr samego `grep` i tak jest zwykle pusty. Strumień błędów `komenda` w ogóle nie wszedł do potoku i poleciał prosto na ekran. Potok łączy wyłącznie stdout lewej strony ze stdin prawej; żeby wciągnąć błędy, `2>&1` musi stać po lewej stronie pionowej kreski.

2. `./skrypt.sh > log.txt 2>&1` i `./skrypt.sh &> log.txt` dają ten sam efekt. Kiedy różnica między nimi ma znaczenie?

Odpowiedź

`&>` jest rozszerzeniem basha (i zsh). Jeśli skrypt startuje z `#!/bin/sh`, a `/bin/sh` to `dash` (Debian, Ubuntu) albo `busybox sh` (kontenery Alpine), `&>` nie zadziała jak przekierowanie — powłoka zinterpretuje `&` jako uruchomienie w tle, a `> log.txt` jako osobne przekierowanie pustej komendy. Efekt bywa cichy i mylący. W skryptach przenośnych zawsze `> plik 2>&1`.

3. Usługa działa, ale `journalctl -u usluga` nie pokazuje żadnych komunikatów o błędach, choć aplikacja wyraźnie ma problem. Jak sprawdzisz, dokąd naprawdę idzie jej stderr?

Odpowiedź

Znajdź PID (`systemctl show -p MainPID --value usluga`) i obejrzyj `ls -l /proc/PID/fd`. Jeśli deskryptor `2` wskazuje na `/dev/null`, ktoś ustawił `StandardError=null` w pliku jednostki albo skrypt startowy sam dopisał `2>/dev/null`. Sprawdź to przez `systemctl cat usluga`. Bywa też, że aplikacja pisze do własnego pliku i wtedy deskryptor 2 wskazuje na ten plik.

4. Dlaczego `find / -name '*.conf' 2>/dev/null | wc -l` daje poprawny wynik, a `find / -name '*.conf' | wc -l 2>/dev/null` sypie ekran komunikatami?

Odpowiedź

W pierwszej wersji `2>/dev/null` należy do `find` i wycisza jego komunikaty `Permission denied`. W drugiej należy do `wc`, który niczego nie zgłasza — a stderr `find` nadal jedzie na terminal. Liczba w obu przypadkach wyjdzie ta sama, bo błędy nigdy nie wchodziły do potoku; różni się tylko czytelność ekranu. To dobra ilustracja zasady, że przekierowanie jest własnością pojedynczej komendy, nie całej linii.

Powiązane: Przekierowania: >, >>, 2>, &>, <, tee · Potoki i filozofia Unix · cat: wypisywanie i sklejanie plików · head i tail: okno na początek i koniec pliku · Kody wyjścia i obsługa błędów w

Źródła: `man 1 bash` (sekcja REDIRECTION) · `man 2 dup2` · `man 3 isatty` · `man 4 null` · `man 4 full` · `man 5 proc` · `man 1 stdbuf` · POSIX.1-2024, Shell Command Language 2.7

Revision #19
Created 2026-09-17 11:56:13 UTC by Tomasz Konieczny
Updated 2026-10-03 19:07:24 UTC by Tomasz Konieczny