Przekierowania: >, >>, 2>, &>, <, tee

Poziom: podstawy **Dotyczy: wszystkie dystrybucje** **Czas: ~14 min**

Przekierowania podpinają strumienie procesu do plików zamiast do terminala. Robi to powłoka, zanim komenda w ogóle wystartuje — i z tej jednej własności bierze się większość pomyłek na tej stronie.

Skróty użyte na tej stronie (1)

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

Fiszka

ZMIENIA SYSTEM

Nadpisanie pliku: >

```
echo "pierwsza wersja" > raport.txt
echo "druga wersja" > raport.txt
cat raport.txt
```

Stdout trafia do pliku, który zostaje obcięty do zera bajtów, zanim cokolwiek się zapisze. Druga komenda kasuje wynik pierwszej bez ostrzeżenia — to jest domyślne zachowanie, nie błąd.

Przykład:

```
druga wersja
```

cat = concatenate: sklej i wypisz pliki

◫◫ ZMIENIA SYSTEM

Dopisanie na koniec: >>

```
echo "trzecia wersja" >> raport.txt
cat raport.txt
```

Dwa znaki zamiast jednego dopisują na koniec pliku zamiast go obcinać. Plik, który nie istnieje, zostaje utworzony od pustego i od razu dopisany.

Przykład:

```
druga wersja
trzecia wersja
```

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

◫◫ ZMIENIA SYSTEM

Stderr do pliku, nadpisanie: 2>

```
ls /etc /nieistnieje 2> bledy.txt
cat bledy.txt
```

Przekierowuje wyłącznie deskryptor 2. Stdout (`ls`) nadal leci na ekran — na terminalu zobaczymy normalną listę `/etc`, a błąd zniknie z ekranu i wyląduje w pliku.

Przykład:

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

◫◫ **ls** = list: wypisz zawartość katalogu · **cat** = concatenate: sklej i wypisz pliki

◫◫ ZMIENIA SYSTEM

Stderr do pliku, dopisanie: 2>>

```
ls /etc /nieistnieje 2>> bledy.txt
cat bledy.txt
```

Kolejne uruchomienie dopisuje drugą linię błędu zamiast nadpisać pierwszą. Przydaje się przy zbieraniu błędów z wielu przebiegów tego samego zadania w jeden plik.

Przykład:

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

ls = list: wypisz zawartość katalogu · **cat** = concatenate: sklej i wypisz pliki

ZMIENIA SYSTEM

Oba strumienie do jednego pliku: &>

```
ls /etc /nieistnieje &> wszystko.txt
head -1 wszystko.txt
```

Bashowy skrót łączący `>` i `2>&1` w jeden operator. Krócej niż wersja POSIX-owa, ale nie zadziała pod `/bin/sh`, jeśli `sh` to `dash` albo `busybox sh`.

Przykład:

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

ls = list: wypisz zawartość katalogu · **head** = pokaż początek pliku, domyślnie dziesięć pierwszych linii

ZMIENIA SYSTEM

Oba strumienie, kolejność ma znaczenie: > plik 2>&1

```
ls /etc /nieistnieje > wszystko.txt 2>&1
cat wszystko.txt
```

To jest poprawna kolejność: najpierw stdout trafia do pliku, potem stderr kopiuje aktualny cel stdout — czyli ten sam plik. Wersja przenośna, POSIX-owa, działa w każdej powłoce.

Przykład:

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

```
adjtime
aliases
alternatives
...
```

Odwrócona kolejność daje inny wynik:

```
ls /etc /nieistnieje 2>&1 > zle.txt
```

Błąd wyleciał na ekran, bo w momencie przetwarzania `2>&1` stdout jeszcze nie był przekierowany:

Przykład:

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

A `zle.txt` zawiera wyłącznie dane, bez ani jednej linii błędu:

```
cat zle.txt
```

Przykład:

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

ls = list: wypisz zawartość katalogu · **cat** = concatenate: sklej i wypisz pliki

TYLKO ODCZYT

Wejście z pliku: <

```
wc -l < /etc/services
```

Powłoka otwiera plik i podaje go komendzie jako deskryptor 0 — jeden proces, zero kopiowania przez potok. `wc` dostaje plik, a nie strumień, więc może po nim skakać zamiast tylko go czytać liniowo.

Przykład:

```
11473
```

wc = word count: policz linie, słowa albo znaki w pliku

ZMIENIA SYSTEM

Here-document: <<

```
cat > /tmp/raport.txt <<EOF
Host: $(hostname)
Data: $(date +%F)
EOF
cat /tmp/raport.txt
```

Wkleja blok tekstu prosto na stdin komendy, bez pliku pośredniego. Bez cudzysłowów wokół `EOF` powłoka podstawia zmienne i wykonuje `$(...)` wewnątrz bloku.

Przykład:

```
Host: srv01
Data: 2026-09-17
```

Z cudzysłowem wokół `EOF` (`<<'EOF'`) tekst idzie dosłownie, bez żadnego podstawiania:

```
cat > /tmp/raport.txt <<'EOF'
Host: $(hostname)
EOF
cat /tmp/raport.txt
```

Przykład:

```
Host: $(hostname)
```

Wariant `<<-EOF` (z myślnikiem) usuwa wiodące tabulatory z każdej linii bloku, co pozwala wciąć here-document wewnątrz funkcji bez psucia treści.

cat = concatenate: sklej i wypisz pliki

TYLKO ODCZYT

Here-string: <<<

```
tr a-z A-Z <<< "tekst"
```

Skrócona forma here-document dla jednej linii — bez linii `EOF`, bez wcięć. Bash dokłada znak nowej linii na końcu automatycznie. To bashizm, w `dash` nie zadziała.

Przykład:

```
TEKST
```

tr = translate: zamień albo usuń znaki w strumieniu tekstu

ZMIENIA SYSTEM

Zapis do pliku i dalej na ekran: tee

```
echo "wpis testowy" | tee dziennik.txt
```

`tee` czyta stdin i pisze go jednocześnie na stdout **i** do podanego pliku — stąd widzimy ten sam tekst na ekranie i w pliku za jednym uruchomieniem.

Przykład:

```
wpis testowy
```

ZMIENIA SYSTEM

Dopisanie zamiast nadpisania: tee -a

```
echo "kolejny wpis" | tee -a dziennik.txt
```

`-a` dopisuje do pliku zamiast go obcinać. Na ekranie i tak widzimy to samo, co wpisaliśmy — `tee` zawsze kopiuje stdin na stdout, niezależnie od flagi.

Przykład:

```
kolejny wpis
```

Dowód, że w pliku obie linie zostały:

```
cat dziennik.txt
```

Przykład:

```
wpis testowy
kolejny wpis
```

cat = concatenate: sklej i wypisz pliki

ZMIENIA SYSTEM

Zapis do pliku wymagającego roota: sudo tee

```
echo "net.ipv4.ip_forward = 1" | sudo tee /etc/sysctl.d/99-local.conf
```

`tee` działa pod `sudo`, więc to **on** otwiera plik, z uprawnieniami roota — nie nasza powłoka. Jeśli nie chcemy echa na ekranie, dokładamy `> /dev/null`.

Przykład:

```
net.ipv4.ip_forward = 1
```

ZMIENIA SYSTEM

Przekierowanie całego bloku poleceń: { } >

```
{ echo "start bloku"; date; } > blok.txt
cat blok.txt
```

Nawiasy klamrowe grupują kilka komend w jednostkę, do której stosuje się jedno przekierowanie. Bez nich `>` dotyczyłoby wyłącznie ostatniej komendy w linii. Składnia jest wybredna: po `{` musi być spacja lub nowa linia, a przed `}` średnik lub nowa linia.

Przykład:

```
start bloku
Thu Sep 17 09:14:11 CEST 2026
```

cat = concatenate: sklej i wypisz pliki

ZMIENIA SYSTEM

Przekierowanie reszty skryptu: exec >

```
#!/bin/bash
exec > /var/log/proba-exec.log 2>&1
echo "Start: $(date)"
```

`exec` bez nazwy programu nie uruchamia niczego nowego — zmienia deskryptory **bieżącej** powłoki. Od tej linii do końca skryptu wszystko, co on i jego dzieci wypiszą, idzie do wskazanego pliku, bez potrzeby dopisywania przekierowania przy każdej komendzie.

```
cat /var/log/proba-exec.log
```

Przykład:

```
Start: Thu Sep 17 09:12:03 CEST 2026
```

exec = zastąp bieżący proces powłoki wskazanym programem, bez tworzenia nowego procesu · **cat** = concatenate: sklej i wypisz pliki

ZMIENIA SYSTEM

Wymuszenie nadpisania mimo `noclobber`: >|

```
set -o noclobber
echo a > chroniony.txt
echo b > chroniony.txt
```

Przy włączonym `noclobber` zwykłe `>` na istniejący plik kończy się błędem zamiast go skasować.

Przykład:

```
bash: chroniony.txt: cannot overwrite existing file
```

Gdy naprawdę chcemy nadpisać, `>|` przełamuje tę ochronę punktowo, bez wyłączania `noclobber` na stałe:

```
echo b >| chroniony.txt
cat chroniony.txt
```

Przykład:

```
b
```

cat = concatenate: sklej i wypisz pliki

ZMIENIA SYSTEM

Zamknięcie deskryptora: 2>&-

```
find /var -name '*.log' 2>&-
```

Zamyka deskryptor 2 zamiast przekierować go w bezpieczne miejsce. Program, który mimo to spróbuje pisać na stderr, dostaje błąd `EBADF` — a część narzędzi reaguje na to źle, łącznie z ryzykiem, że kolejny otwierany przez program plik dostanie numer 2 jako pierwszy wolny deskryptor. Pełne wyjaśnienie tego ryzyka w sekcji *Częste błędy* niżej.

Zapis	Co robi
`> plik`	stdout do pliku, obcina istniejący
`>> plik`	stdout do pliku, dopisuje na koniec
`2> plik`	stderr do pliku, obcina
`2>> plik`	stderr do pliku, dopisuje
`> plik 2>&1`	oba strumienie do pliku (POSIX)
`&> plik`, `>& plik`	to samo, bashizm
`< plik`	plik jako stdin
`<< EOF`	here-document — blok tekstu na stdin
`<<'EOF'`	here-document bez podstawiania zmiennych
`<<- EOF`	here-document ignorujący wiodące tabulatory
`<<< "tekst"`	here-string — jedna linia na stdin

Zapis	Co robi
`>	plik`
`2>&-`	zamknięcie deskryptora 2
`exec > plik`	przekierowanie reszty skryptu

nadpisaniem mo `noclobber`

Uważaj

- `>` obcina plik **przed** uruchomieniem komendy, więc `sort plik > plik` zostawia pusty plik.
- `sudo komenda > /etc/plik` nie działa — przekierowanie wykonuje Twoja powłoka, nie `sudo`.
- `>>` nie jest bezpieczne przy równoległych zapisach powyżej `PIPE_BUF` (4096 bajtów na Linuksie) — linie potrafią się posiekać.

POSIX = Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach · **find** = przeszukaj drzewo katalogów według nazwy, typu, daty albo rozmiaru

Z życia wzięte

Z ŻYCIA WZIĘTE

> Wtorek, 15:50, ostatnie dni zamknięcia miesiąca. Tomek chce odświeżyć raport i wpisuje z literówką `eksport dane.csv > raport.txt` na `sql01`. Powłoka od razu zgłasza `command not found`, ale `raport.txt` i tak istnieje i ma zero bajtów, bo plik został otwarty z `O_TRUNC`, zanim ktokolwiek sprawdził, czy program w ogóle istnieje. Stary raport z piątku właśnie zniknął, więc Tomek uczy się na pamięć, że przekierowanie działa niezależnie od tego, czy komenda wystartuje.

Po co to istnieje

Komenda nie wie, dokąd pisze. `ls` zapisuje bajty do deskryptora numer 1 i to jest cała jego wiedza na ten temat — czy po drugiej stronie jest terminal, plik na dysku, potok, czy socket sieciowy, jest sprawą jądra i powłoki.

Ta niewiedza jest celowa. Dzięki niej ten sam `ls` obsługuje cztery różne scenariusze, nie mając ani jednej linijki kodu o plikach wyjściowych:

```
ls /etc                  # na ekran
ls /etc > lista.txt      # do pliku
ls /etc | wc -l          # do innego programu
ls /etc | ssh srv01 'cat > /tmp/lista'   # przez sieć
```

Autor `ls` nie musiał przewidzieć żadnego z nich. Przekierowanie jest warstwą pomiędzy: powłoka ustawia deskryptory procesu potomnego **zanim** wywoła `exec`, i od tego momentu program pisze tam, gdzie mu pokazano.

Analogia

Przekierowanie to **zwrotnica kolejowa**.

Pociąg jedzie po torze i nie ma żadnego wpływu na to, którym rozgałęzieniem pojedzie. Nie zna rozkładu, nie widzi, co jest za zakrętem. Maszynista po prostu prowadzi skład do przodu. O kierunku decyduje dróżnik, który **przed** przyjazdem pociągu przekłada zwrotnicę. Gdy skład wjeżdża na rozjazd, wszystko jest już postanowione.

`>` to ten dróżnik. Powłoka przekłada zwrotnicę — otwiera plik, podpina go pod deskryptor 1 — a dopiero potem puszcza pociąg, czyli uruchamia komendę. Program jedzie tam, gdzie tor jest ustawiony, i nie jest w stanie zauważyć różnicy.

Z tego wynika pierwsze ograniczenie, i to jest dokładnie ta pułapka, która na tej stronie zabija najwięcej plików. **Dróżnik działa pierwszy.** Jeśli przygotowanie toru wymaga oczyszczenia

bocznicy, bocznica zostaje oczyszczona, zanim pociąg w ogóle ruszył. Powłoka, otwierając plik do zapisu, obcina go do zera bajtów — natychmiast, bez pytania. Jeśli ten sam plik miał być jednocześnie źródłem danych dla komendy, komenda zastanie pustą bocznicę. Stąd `cat plik > plik` i `sort plik > plik` niszczą plik.

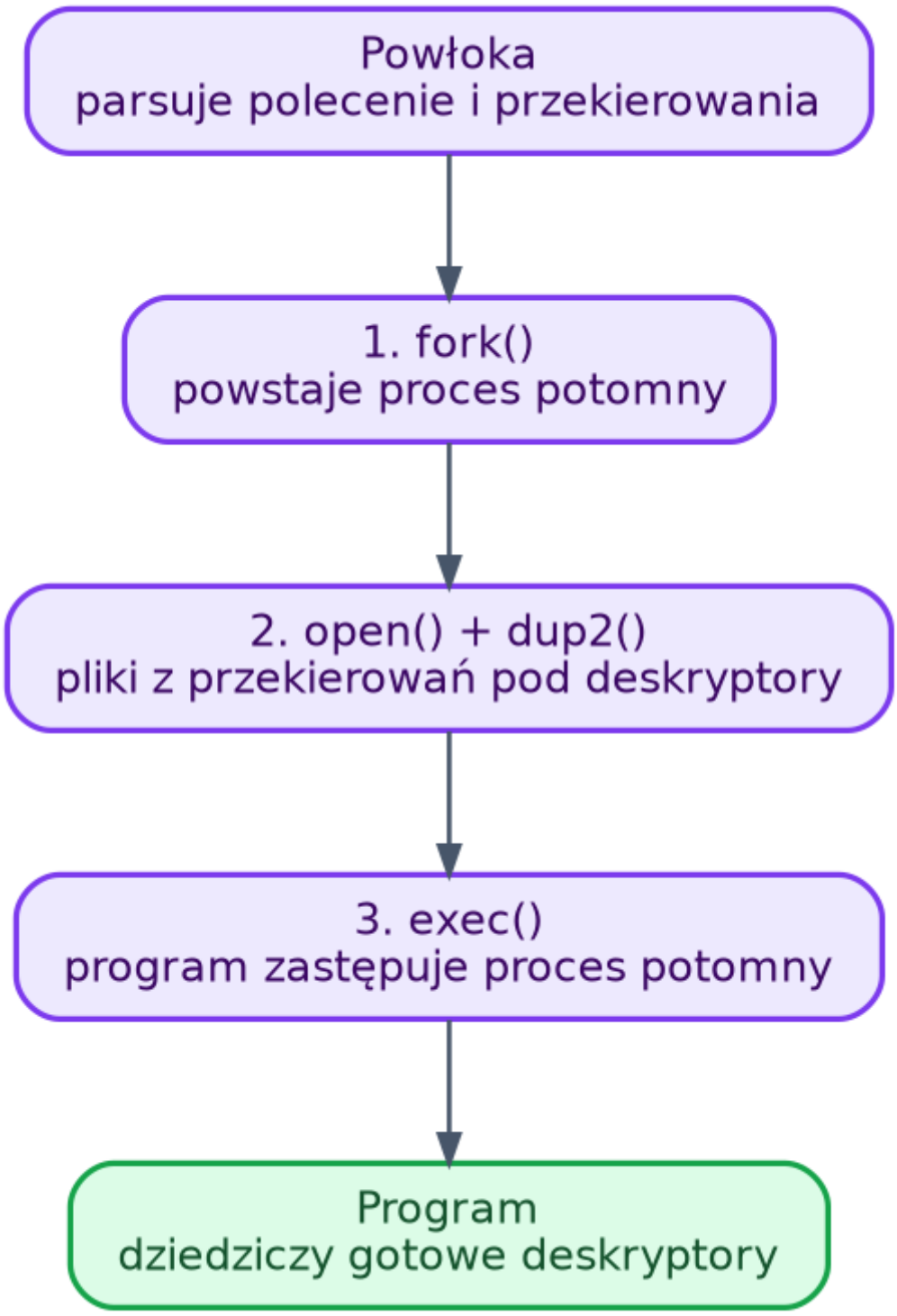
Drugie ograniczenie: **dróżnik pracuje dla swojej stacji**. Zwrotnicę przekłada dróżnik na stacji, z której pociąg wyrusza, a nie maszynista i nie stacja docelowa. Kiedy piszesz `sudo echo x > /etc/plik`, zwrotnicę ustawia Twoja powłoka, działająca na Twoich uprawnieniach. `sudo` dotyczy wyłącznie pociągu — czyli `echo`. Dlatego dostajesz `Permission denied`, mimo że komenda była pod `sudo`.

Jak to działa

Rozwiń: jak to działa pod spodem

Powłoka rozbiera linię komendy na trzy części: nazwę programu, argumenty i **przekierowania**. Przekierowania nie trafiają do programu jako argumenty — program ich nigdy nie widzi. Kolejność kroków przy uruchomieniu:

1. Powłoka robi `fork()` — powstaje proces potomny.
2. W procesie potomnym powłoka **otwiera pliki z przekierowań** i podpina je pod odpowiednie deskryptory (`open()` + `dup2()`).
3. Dopiero teraz woła `exec()` — program zastępuje sobą proces potomny i dziedziczy ustawione deskryptory.



Trzy kroki przekierowania: fork, open i dup2, dopiero potem exec programu (kliknięcie otwiera pełny rozmiar)

Krok 2 jest tutaj kluczowy. Otwarcie pliku przez `>` używa flag `O_WRONLY | O_CREAT | O_TRUNC`. `O_TRUNC` znaczy „obetnij do zera bajtów". Dzieje się to natychmiast, niezależnie od tego, czy

komenda w ogóle się uruchomi — a nawet jeśli nazwa programu jest literówką:

```
lss /etc > lista.txt
```

```
bash: lss: command not found
```

Plik `lista.txt` istnieje i ma zero bajtów. Powłoka utworzyła go, zanim zorientowała się, że `lss` nie istnieje.

`>>` używa `O_APPEND` zamiast `O_TRUNC`. Poza brakiem obcięcia daje to jeszcze jedną własność: jądro atomowo przesuwa każdy zapis na aktualny koniec pliku, więc dwa procesy dopisujące do tego samego logu nie nadpisują się nawzajem. Gwarancja obowiązuje jednak tylko do `PIPE_BUF` (4096 bajtów na Linuksie) — dłuższe zapisy potrafią się przepleść.

Wejście: `<` kontra `cat plik |`

Te dwie komendy dają ten sam wynik:

```
wc -l < /etc/services
cat /etc/services | wc -l
```

```
11473
11473
```

Różnią się jednak zasadniczo. W pierwszej powłoka otwiera plik i podaje go `wc` jako deskryptor 0 — jeden proces, zero kopiowania, a `wc` dostaje **plik**, więc może po nim skakać przez `lseek()`. W drugiej startują dwa procesy, a dane przechodzą przez bufor potoku w jądrze; `wc` dostaje **strumień**, po którym cofnąć się nie da.

Dla `wc` to różnica głównie wydajnościowa. Dla `tail -n 5` to różnica między odpowiedzią natychmiastową a przemieleniem całego pliku. Trzecia forma, `wc -l /etc/services`, jest jeszcze inna: `wc` sam otwiera plik i zna jego nazwę, więc wypisze ją obok liczby.

Here-document i here-string

> **Pojęcie: here-document.** Blok tekstu wpisany wprost w kodzie skryptu albo w linii komend, który powłoka podaje komendzie jako standardowe wejście — bez tworzenia pliku pośredniego. Zaczyna się od `<<SŁOWO` (najczęściej `<<EOF`), a kończy linią

> zawierającą samo `SŁOWO`. Wszystko pomiędzy trafia na stdin komendy dosłownie, tak jakby ktoś to wpisał ręcznie z klawiatury.

```
cat > /tmp/raport.txt <<EOF
Host: $(hostname)
Data: $(date +%F)
Użytkownik: $USER
EOF
```

```
Host: srv01
Data: 2026-09-17
Użytkownik: adm_user
```

Bez cudzysłowów wokół `EOF` powłoka podstawia zmienne i wykonuje `$(...)` wewnątrz bloku. Z cudzysłowami (`<<'EOF'`) tekst idzie dosłownie:

```
cat > /tmp/raport.txt <<'EOF'
Host: $(hostname)
EOF
```

```
Host: $(hostname)
```

To rozróżnienie decyduje o poprawności każdego skryptu, który generuje inny skrypt albo plik konfiguracyjny ze znakami `$`. Reguła: **jeśli nie potrzebujesz podstawiania, zawsze cytuj `EOF`**.

Wariant `<<-EOF` usuwa wiodące **tabulatory** (nie spacje) z każdej linii bloku, łącznie z linią zamykającą — pozwala wciąć here-document wewnątrz funkcji bez psucia treści.

Here-string `<<<` to skrócona forma dla jednej linii:

```
tr a-z A-Z <<< "srv01 srv02"
```

```
SRV01 SRV02
```

Bash dokłada znak nowej linii na końcu. `<<<` jest bashizmem — w `dash` nie zadziała.

tee

`tee` czyta stdin i pisze go jednocześnie na stdout **i** do podanych plików. Nazwa pochodzi od trójnika hydraulicznego, litery T: jedno wejście, dwa wyjścia.

```
./deploy.sh 2>&1 | tee /var/log/deploy-$(date +%F).log
```

Widzisz wszystko na ekranie i masz to w pliku. `tee -a` dopisuje zamiast nadpisywać, `tee` przyjmuje wiele plików naraz i można go wstawić w środek potoku:

```
dmesg | tee /tmp/pelny.log | grep -i error
```

Pełne wyjście trafia do pliku, a dalej przez potok jadą tylko dopasowania.

Krok po kroku

1. Rozdziel dane od błędów w jednym poleceniu.

```
cd /tmp
ls /etc /nieistnieje > dane.txt 2> bledy.txt
cat bledy.txt
wc -l dane.txt bledy.txt
```

```
ls: cannot access '/nieistnieje': No such file or directory
     238 dane.txt
       1 bledy.txt
```

Dwa niezależne przekierowania w jednej linii, każde dotyczy innego deskryptora. `dane.txt` nie zawiera ani jednej linii błędu — sprawdź to przez `grep -c cannot dane.txt`, wynik będzie `0`.

2. Zapisz plik w katalogu wymagającym roota, tym razem z dopisaniem.

To jest ten moment, w którym większość ludzi po raz pierwszy trafia na ścianę uprawnień:

```
echo "203.0.113.10 storage01" | sudo tee -a /etc/hosts > /dev/null
tail -1 /etc/hosts
```

```
203.0.113.10 storage01
```

`tee` działa pod `sudo`, więc to **on** otwiera `/etc/hosts` z uprawnieniami roota, a `-a` dopisuje zamiast nadpisywać. `> /dev/null` na końcu wycisza echo na ekranie — bez niego zobaczyłbyś tę samą linię

dwa razy, raz od `tee`, raz od `tail`.

3. Zbierz wyjście z całego bloku.

```
{
  echo "=== raport $(date +%F) ==="
  df -h /
  echo "--- pamięć ---"
  free -h
} > /tmp/raport.txt
```

Nawiasy klamrowe grupują komendy w jednostkę, do której stosuje się jedno przekierowanie. Bez nich `>` dotyczyłoby wyłącznie ostatniej komendy. Składnia jest wybredna: po `{` musi być spacja lub nowa linia, a przed `}` średnik lub nowa linia.

To samo działa dla pętli:

```
for h in srv01 srv02 storage01; do
  echo "$h: $(getent hosts "$h" | awk '{print $1}')"
done > /tmp/adresy.txt
```

4. Przekieruj cały skrypt.

```
#!/bin/bash
exec > /var/log/moj-skrypt.log 2>&1

echo "Start: $(date)"
./etap1.sh
./etap2.sh
```

`exec` bez nazwy programu nie uruchamia niczego — zmienia deskryptory **bieżącej** powłoki. Od tej linii do końca skryptu wszystko, co wypiszą skrypt i jego dzieci, idzie do logu. To najczystszy sposób na logowanie zadania z crona.

Jeśli chcesz mieć i log, i ekran:

```
exec > >(tee -a /var/log/moj-skrypt.log) 2>&1
```

5. Sprawdź, czego `noclobber` naprawdę pilnuje.

```
set -o noclobber
echo "wpis 1" >> dziennik-audytu.txt
cat dziennik-audytu.txt
```

```
echo "wpis 2" > dziennik-audytu.txt
```

```
wpis 1
bash: dziennik-audytu.txt: cannot overwrite existing file
```

Dopisanie przez `>>` przeszło bez problemu, mimo że `noclobber` był włączony — ochrona dotyczy wyłącznie `>` na istniejący plik, nie `>>`. Dopiero próba nadpisania przez `>` została zablokowana. Wyłącz ochronę przez `set +o noclobber`, gdy skończysz eksperymentować.

Częste błędy

`sudo echo "wartosc" > /etc/plik` — klasyka nad klasykami.

```
sudo echo "net.ipv4.ip_forward = 1" > /etc/sysctl.d/99-local.conf
```

```
bash: /etc/sysctl.d/99-local.conf: Permission denied
```

Jesteś pewien, że to błąd `sudo`, więc próbujesz `sudo -i`, `sudo su -`, sprawdzasz `sudoers`. Tymczasem `sudo` w ogóle nie brało w tym udziału. Powłoka rozbiera linię **zanim** uruchomi cokolwiek: widzi `>`, próbuje otworzyć plik — na Twoich uprawnieniach, bo to Twoja powłoka — i dostaje odmowę. `sudo echo` nigdy nie wystartowało.

Trzy poprawne rozwiązania:

```
echo "net.ipv4.ip_forward = 1" | sudo tee /etc/sysctl.d/99-local.conf > /dev/null
sudo sh -c 'echo "net.ipv4.ip_forward = 1" > /etc/sysctl.d/99-local.conf'
sudo tee /etc/sysctl.d/99-local.conf > /dev/null <<'EOF'
net.ipv4.ip_forward = 1
EOF
```

Pierwsze jest najczęstsze i najbezpieczniejsze. Drugie działa, ale uruchamia całą powłokę z uprawnieniami roota — a jeśli wstawisz tam zmienną, wpuszczasz jej zawartość do kodu wykonywanego jako root. Trzecie jest najlepsze dla plików wielolinijkowych.

`cat plik > plik` — kasowanie pliku, który chciałeś tylko przetworzyć.

```
sort /tmp/lista.txt > /tmp/lista.txt
wc -l /tmp/lista.txt
```

```
0 /tmp/lista.txt
```

Nie ma błędu, nie ma ostrzeżenia. Powłoka otworzyła `lista.txt` z `O_TRUNC` i obcięła go do zera, zanim `sort` zdążył go otworzyć do odczytu. `sort` przeczytał pusty plik i zapisał pustkę.

Dotyczy wszystkiego: `grep wzorzec plik > plik`, `tr -d '\r' < plik > plik`, `uniq plik > plik`. Rozwiązania:

```
sed -i 's/stare/nowe/' plik.txt           # sed umie w miejscu
sort plik.txt -o plik.txt                 # sort ma własną flagę
sort plik.txt | sponge plik.txt           # moreutils
sort plik.txt > plik.tmp && mv plik.tmp plik.txt   # uniwersalne
```

Ostatnia forma z `&&` jest najbezpieczniejsza: jeśli `sort` zawiedzie, `mv` się nie wykona i oryginał zostaje.

Zła kolejność `2>&1`.

```
./backup.sh 2>&1 > /var/log/backup.log
```

Plik powstaje, dane w nim są, ale komunikaty o błędach nadal lecą na terminal albo do maila z crona. Przekierowania idą od lewej do prawej: `2>&1` skopiowało wskaźnik jedynki **zanim** jedynkę przestawiono na plik, więc dwójka wskazuje na terminal.

Poprawnie — `2>&1` zawsze po przekierowaniu stdout:

```
./backup.sh > /var/log/backup.log 2>&1
```

Dobra wiadomość: przy `&>` pomyłka jest niemożliwa, bo to jeden operator. Zła: `&>` nie jest POSIX-owe.

`>>` w skrypcie uruchamianym wielokrotnie, gdzie miało być `>`.

```
0 */6 * * * /opt/skrypty/inwentarz.sh >> /var/tmp/inwentarz.csv
```

Po miesiącu plik ma 120 kopii tych samych danych sklejonych w jeden ciąg, a import do bazy tworzy 120 razy te same rekordy. `>>` nigdy nie zgłosi problemu — będzie rósł, aż zapełni partycję. Odwrotna pomyłka jest równie kosztowna: `>` tam, gdzie miało być `>>`, i log nadpisywany przy każdym uruchomieniu, więc historii nie ma wcale.

Ucieczka od `2>&-` zamiast `2>/dev/null`.

```
find /var -name '*.log' 2>&-
```

Wygląda elegancko: zamykasz deskryptor zamiast marnować `/dev/null`. Problem w tym, że program pisząc do zamkniętego deskryptora dostaje `EBADF` i część narzędzi reaguje na to źle —

kończy pracę, wypisuje własny błąd albo, co gorsza, otwierając nowy plik dostaje numer 2 jako pierwszy wolny deskryptor i zaczyna wysypywać komunikaty diagnostyczne do środka tego pliku. `2>/dev/null` zostawia deskryptor otwarty i poprawny. Zamykanie deskryptorów zostaw do sytuacji, w których dokładnie wiesz, po co to robisz.

□□ 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 — zobacz obcięcie na własne oczy

```
cd /tmp
mkdir -p cw-przekierowania && cd cw-przekierowania
seq 1 5 > liczby.txt
wc -l liczby.txt
sort -r liczby.txt > liczby.txt
wc -l liczby.txt
```

Odpowiedź

```
5 liczby.txt
0 liczby.txt
```

Plik zniknął w sensie zawartości. To samo zrobi zwykła literówka — `sortt liczby.txt > liczby.txt` obcina plik, zanim powłoka zdąży stwierdzić, że `sortt` nie istnieje. Odtwórz plik i zrób to poprawnie:

```
seq 1 5 > liczby.txt
sort -r liczby.txt -o liczby.txt
cat liczby.txt
```

```
5
4
```

```
3
2
1
```

Ćwiczenie 2 — udowodnij, że kolejność `2>&1` ma znaczenie

```
cd /tmp/cw-przekierowania
ls /etc /nieistnieje > A.txt 2>&1
ls /etc /nieistnieje 2>&1 > B.txt
grep -c "No such file" A.txt B.txt
```

Odpowiedź

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

W drugim przypadku komunikat pojawił się na terminalu między komendami. Zapamiętaj kolejność: **najpierw dokąd, potem co**.

Ćwiczenie 3 — here-document z cytowaniem i bez

```
cd /tmp/cw-przekierowania
cat > z-podstawieniem.txt <<EOF
user=$USER
cena=$100
EOF

cat > bez-podstawienia.txt <<'EOF'
user=$USER
cena=$100
EOF
```

```
cat z-podstawieniem.txt
echo '---'
cat bez-podstawienia.txt
```

Odpowiedź

```
user=adm_user
cena=00
---
user=$USER
cena=$100
```

`$100` w wersji bez cytowania zostało zinterpretowane jako `$1` (pierwszy argument, tu pusty) plus `00`. To jest typowy sposób, w jaki generowany plik konfiguracyjny z hasłem albo kwotą cicho traci fragment treści. Gdy blok zawiera `$`, cytuj `EOF`.

Pytania kontrolne

1. `sudo echo "test" >> /etc/hosts` kończy się `Permission denied`. `echo "test" | sudo tee -a /etc/hosts` działa. Co dokładnie robi różnicę?

Odpowiedź

To, **kto otwiera plik**. W pierwszej wersji otwiera go Twoja powłoka, która nie ma uprawnień do `/etc/hosts` — i robi to zanim `sudo` w ogóle wystartuje. W drugiej plik otwiera `tee`, a `tee` jest uruchomiony przez `sudo`, więc działa jako root. Przekierowanie jest zawsze operacją powłoki, która je zobaczyła w linii komendy, i żadne `sudo` przed komendą tego nie zmieni.

2. Chcesz usunąć puste linie z `/tmp/dane.csv` i zapisać wynik w tym samym pliku. Które z tych rozwiązań są bezpieczne: `grep -v '^$' dane.csv > dane.csv`, `sed -i '/^$/d' dane.csv`, `grep -v '^$' dane.csv > tmp && mv tmp dane.csv`?

Odpowiedź

Pierwsze niszczy plik — powłoka obcina `dane.csv` przed startem `grep`. Drugie i trzecie są poprawne. `sed -i` w rzeczywistości też pisze do pliku tymczasowego i podmienia go na końcu, ale robi to sam, więc oryginał żyje aż do momentu podmiany. Trzecie jest uniwersalne i dodatkowo bezpieczne dzięki `&&`: jeśli `grep` zawiedzie, `mv` się nie wykona.

3. Skrypt zaczyna się od `exec >> /var/log/skrypt.log 2>&1`. Co się stanie, jeśli skrypt uruchomisz ręcznie z terminala i dołożysz na końcu `| tee ekran.txt`?

Odpowiedź

`tee` nie zobaczy nic i `ekran.txt` będzie pusty. Potok podpina stdout skryptu do `tee`, ale pierwsza linia skryptu natychmiast przestawia deskryptor 1 na plik logu — od tego momentu potok jest odłączony i nikt do niego nie pisze. `exec` wewnątrz skryptu wygrywa z przekierowaniem ustawionym z zewnątrz, bo działa później. Jeśli chcesz obu rzeczy naraz, użyj w skrypcie `exec > >(tee -a /var/log/skrypt.log) 2>&1`.

4. Dwa zadania z crona dopisują do tego samego pliku przez `>>` i czasem w logu pojawiają się poszarpane linie. Skoro `O_APPEND` jest atomowe, skąd to się bierze?

Odpowiedź

`O_APPEND` gwarantuje, że pozycja zapisu zostanie ustawiona na koniec pliku atomowo względem samego zapisu — ale niepodzielność całego `write()` jest gwarantowana tylko do rozmiaru `PIPE_BUF`, czyli 4096 bajtów na Linuksie. Dłuższy zapis jądro może rozbić na kilka operacji, a między nimi wejdzie drugi proces. Drugi powód jest częstszy: jeśli proces wypuszcza log przez bibliotekę buforującą, jeden `write()` może zawierać kilka linii naraz, a przeplot następuje na granicy buforów, nie linii. Rozwiązanie: osobny plik na zadanie albo logowanie przez `logger` /journald, które serializuje zapisy.

Powiązane: Trzy strumienie: stdin, stdout, stderr · Potoki i filozofia Unix · cat: wypisywanie i sklejanie plików · Zmienne i cytowanie w bashu · sudo — uprawnienia i pułapki · Logowanie ze skryptów: logger i journald

Źródła: `man 1 bash` (sekcja REDIRECTION) · `man 1 tee` · `man 2 open` (`O_TRUNC`, `O_APPEND`) · `man 2 write` (BUGS, atomowość) · `man 7 pipe` · `man 1 sponge` (moreutils) · POSIX.1-2024, Shell Command Language 2.7

Revision #19
Created 2026-09-17 11:56:15 UTC by Tomasz Konieczny
Updated 2026-10-03 19:07:26 UTC by Tomasz Konieczny