Obsługa błędów, trap i sprzątanie

Poziom: średniozaawansowany **Dotyczy: bash 4+/5+** **Czas: ~22 min**

Tryb ścisły zatrzymuje skrypt przy błędzie, ale sam nic nie sprząta i niczego nie loguje. Ta strona dodaje resztę: `trap` do reagowania na sygnały i wyjście ze skryptu, `mktemp` do bezpiecznych plików tymczasowych, funkcję `die` do czytelnych komunikatów i `flock` do ochrony przed równoległym uruchomieniem.

Skróty użyte na tej stronie (4)

Skrót	Co znaczy
PID	Process ID: numer procesu w systemie
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach
SIGTERM	sygnał numer 15: prośba do procesu, żeby zakończył się grzecznie, proces może ją zignorować albo posprzątać przed wyjściem
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera

Fiszka

OSTROŻNIE · NIEODWRACALNE

Sprzątanie przy każdym zakończeniu skryptu: trap ... EXIT

```
katalog_tymczasowy=$(mktemp -d)

sprzataj() {
    rm -rf -- "$katalog_tymczasowy"
}
trap sprzataj EXIT
```

```
echo "pracuję w: $katalog_tymczasowy"
```

`trap` rejestruje komendę albo funkcję, która ma się wykonać w reakcji na sygnał albo na zdarzenie powłoki. `EXIT` nie jest prawdziwym sygnałem systemowym — to zdarzenie samej powłoki, uruchamiane zawsze, gdy skrypt kończy pracę, niezależnie od tego, czy zakończył się sukcesem, przez `exit`, przez `set -e`, czy przez sygnał. Rejestrujemy pułapkę zaraz po utworzeniu zasobu, żeby między utworzeniem a rejestracją nie było okna, w którym przerwanie zostawiłoby zasób bez sprzątania.

rm = remove: usuń (bez kosza, na zawsze) · **trap** = przechwyć sygnał w skrypcie i uruchom w reakcji własny fragment kodu

ZMIENIA SYSTEM

Reakcja na błąd polecenia: trap ... ERR

```
set -eE
trap 'echo "błąd w linii $LINENO, kod $?" >&2' ERR
```

`ERR` uruchamia się w tych samych sytuacjach, w których zadziałałby `set -e` — czyli **nie** w warunku `if`, nie po `&&`/`||`. Flaga `-E` (`errtrace`) jest tu konieczna, żeby pułapka `ERR` dziedziczyła się także do funkcji i podpowłok — bez niej `trap` na `ERR` zdefiniowany na najwyższym poziomie skryptu nie uruchomi się dla błędu wewnątrz funkcji.

trap = przechwyć sygnał w skrypcie i uruchom w reakcji własny fragment kodu

ZMIENIA SYSTEM

Reakcja na Ctrl+C: trap ... INT

```
trap 'echo "przerwano przez użytkownika" >&2; exit 130' INT
```

`INT` to sygnał wysyłany przez `Ctrl+C` w terminalu. Bez własnej obsługi bash i tak przerywa skrypt, ale własna pułapka pozwala dodatkowo posprzątać albo zostawić czytelny komunikat zamiast gołego zerwania w połowie linii.

trap = przechwyć sygnał w skrypcie i uruchom w reakcji własny fragment kodu

ZMIENIA SYSTEM

Reakcja na żądanie zatrzymania: trap ... TERM

```
trap 'echo "otrzymano SIGTERM, kończę porządnie" >&2; exit 143' TERM
```

`TERM` to domyślny sygnał `kill` i `systemctl stop` — sygnał grzecznej prośby o zakończenie, w odróżnieniu od `SIGKILL`, którego przechwycić się nie da. Usługa uruchamiana przez systemd powinna reagować na `TERM`, jeśli ma zdążyć dokończyć bieżącą operację przed wymuszonym zabiciem po `TimeoutStopSec`.

SIGKILL = sygnał numer 9: każe jądru natychmiast zabić proces, bez szansy na sprzątnięcie po sobie · **trap** = przechwyć sygnał w skrypcie i uruchom w reakcji własny fragment kodu · **SIGTERM** = sygnał numer 15: prośba do procesu, żeby zakończył się grzecznie, proces może ją zignorować albo posprzątać przed wyjściem

TYLKO ODCZYT

Reakcja na zerwanie sesji: trap ... HUP

```
trap '' HUP
```

`HUP` historycznie oznaczał zerwanie połączenia z terminalem — dziś najczęściej widujemy go, gdy sesja SSH się rozłącza. Pusty ciąg jako komenda (`trap '' HUP`) oznacza jawne zignorowanie sygnału — dokładnie to, co robi polecenie `nohup` dla procesu uruchamianego w tle.

SSH = Secure Shell: szyfrowane zdalne logowanie do serwera · **trap** = przechwyć sygnał w skrypcie i uruchom w reakcji własny fragment kodu

TYLKO ODCZYT

Wszystkie zdarzenia i sygnały łapane przez trap

Zdarzenie	Rodzaj	Typowe źródło	Typowe zastosowanie w skrypcie
`EXIT`	pseudosygnał powłoki	każde zakończenie skryptu, jakiekolwiek	sprzątanie plików tymczasowych, zawsze
`ERR`	pseudosygnał powłoki	niezerowy kod komendy pod `set -eE`	log z numerem linii i kodem błędu
`INT`	sygnał (2)	`Ctrl+C` w terminalu	komunikat, bezpieczne przerwanie
`TERM`	sygnał (15)	`kill`, `systemctl stop`	dokończenie bieżącej operacji, potem wyjście

Zdarzenie	Rodzaj	Typowe źródło	Typowe zastosowanie w skrypcie
`HUP`	sygnał (1)	zamknięcie terminala, rozłączenie SSH	zignorowanie albo przeładowanie konfiguracji

Uważaj

- `SIGKILL` (9) i `SIGSTOP` (19) nie da się przechwycić żadnym `trap` — jądro obsługuje je bezpośrednio, bez udziału procesu.
- `trap` zarejestrowany wewnątrz funkcji bez `-E` przy `set -e` nie zawsze dziedziczy się tak, jak oczekujemy — testujemy to jawnie, zamiast zakładać.
- Kolejne wywołanie `trap ... EXIT` **nadpisuje** poprzednie, nie dodaje się do niego — jeśli potrzebujemy kilku działań przy wyjściu, wołamy je z jednej funkcji `cleanup`, a nie z kilku osobnych `trap`.

TYLKO ODCZYT

Utworzenie bezpiecznego pliku tymczasowego: mktemp

```
plik_tymczasowy=$(mktemp)
echo "pracuję w pliku: $plik_tymczasowy"
```

`mktemp` tworzy plik o unikalnej, nieprzewidywalnej nazwie i zwraca jego ścieżkę — w przeciwieństwie do ręcznie sklejanej nazwy typu `/tmp/moj_skrypt_$$`, nie da się jej odgadnąć z wyprzedzeniem ani podmienić zawartości przez inny proces w oknie między sprawdzeniem a użyciem.

OSTROŻNIE · NIEODWRACALNE

Utworzenie bezpiecznego katalogu tymczasowego: mktemp -d

```
katalog_tymczasowy=$(mktemp -d)
trap 'rm -rf -- "$katalog_tymczasowy"' EXIT
```

`-d` tworzy katalog zamiast pliku — przydatne, gdy operacja generuje kilka plików pośrednich naraz i chcemy je wszystkie posprzątać jednym `rm -rf` na końcu.

Przykład:

```
/tmp/tmp.X7fQmK2pLr
```

trap = przechwyć sygnał w skrypcie i uruchom w reakcji własny fragment kodu

ZMIENIA SYSTEM

Zakończenie z komunikatem błędu i kodem wyjścia: die

```
die() {
    echo "BŁĄD: $1" >&2
    exit "${2:-1}"
}

[[ -f /etc/myapp.conf ]] || die "brak pliku konfiguracyjnego /etc/myapp.conf" 2
```

Jedna funkcja zamiast powtarzania `echo ... >&2; exit N` w każdym miejscu skryptu, gdzie coś może pójść nie tak. Komunikat idzie na stderr, żeby nie mieszał się ze standardowym wyjściem, gdyby ktoś przekierował skrypt do pliku albo do innego programu.

exit = zakończ bieżącą powłokę albo skrypt z podanym kodem wyjścia

TYLKO ODCZYT

Logowanie ze znacznikiem czasu: log

```
log() {
    printf '[%s] %s\n' "$(date '+%Y-%m-%d %H:%M:%S')" "$1"
}

log "start przetwarzania"
```

Znacznik czasu przy każdej linii logu jest tym, czego brakuje najbardziej, gdy trzeba odtworzyć, ile trwał krok, który padł wczoraj o trzeciej nad ranem.

Przykład:

```
[2026-09-25 03:12:07] start przetwarzania
```

printf = wypisz sformatowany tekst, ściślejsza odmiana echo

◫◫ ZMIENIA SYSTEM

Przekierowanie całego skryptu do pliku logu: exec > >(tee -a plik)

```
exec > >(tee -a /var/log/NazwaUslugi/przebieg.log) 2>&1
echo "ten komunikat trafia jednocześnie na ekran i do pliku"
```

`exec` bez podanej komendy zmienia przekierowania **bieżącej powłoki** na stałe, zamiast tylko jednej komendy — od tej linii do końca skryptu każde standardowe wyjście i błąd lecą przez `tee`, które jednocześnie wypisuje je na ekran i dopisuje do pliku. Podstawienie procesu `>(...)` tworzy w tle proces czytający ze swojego wejścia jak z pliku.

◫◫ **exec** = zastąp bieżący proces powłoki wskazanym programem, bez tworzenia nowego procesu

◫◫ TYLKO ODCZYT

Zapis komunikatu do dziennika systemowego: logger

```
logger -t NazwaUslugi "zadanie zakończone sukcesem"
```

`logger` wysyła pojedynczy komunikat do dziennika systemowego (journald na systemach z systemd, `/var/log/syslog` albo `/var/log/messages` w starszych konfiguracjach) — przydatne dla skryptów cyklicznych, których wynik chcemy widzieć w tym samym miejscu co logi usług systemowych, bez utrzymywania osobnego pliku.

◫◫ **logger** = Logger: zapisuje komunikat w dzienniku systemowym

◫◫ ZMIENIA SYSTEM

Blokada przed równoległym uruchomieniem: flock

```
#!/bin/bash
exec 200>/var/run/NazwaUslugi.lock
flock -n 200 || { echo "inna instancja już działa" >&2; exit 1; }

echo "jedyna aktywna instancja, pracuję"
```

Otwieramy deskryptor 200 na plik blokady, a `flock -n` próbuje nałożyć blokadę bez czekania (`-n`, *non-blocking*) — jeśli inny proces już ją trzyma, `flock` od razu zwraca niepowodzenie zamiast czekać w nieskończoność. Blokada trwa dokładnie tyle, ile żyje deskryptor 200, czyli do końca skryptu — nie trzeba jej ręcznie zwalniać.

□□ **exec** = zastąp bieżący proces powłoki wskazanym programem, bez tworzenia nowego procesu · **flock** = zablokuj plik na czas trwania polecenia, żeby dwa uruchomienia skryptu się nie nałożyły

□□ ZMIENIA SYSTEM

Ponowienie próby z odczekaniem: retry

```
proba=0
maksymalna_liczba_prob=5
do_zrobienia() { curl -fsS https://api.zielona.example/status; }

until do_zrobienia; do
    proba=$((proba + 1))
    if (( proba >= maksymalna_liczba_prob )); then
        echo "nie udało się po $maksymalna_liczba_prob próbach" >&2
        exit 1
    fi
    echo "próba $proba nieudana, czekam $((proba * 2))s"
    sleep $((proba * 2))
done
```

Odczekanie rosnące z każdą próbą (*exponential backoff* w uproszczonej, liniowej wersji) zamiast stałego interwału zmniejsza obciążenie usługi, która akurat ma przejściowy problem, zamiast dobijać się do niej co sekundę.

> **Uważaj**
>
> - `flock` na pliku, a nie na deskryptorze otwartym przez `exec`, zwalnia blokadę zaraz po zakończeniu komendy `flock`, a nie po zakończeniu skryptu — dlatego wzorzec z `exec 200>plik` jest tym właściwym do trzymania blokady przez cały czas trwania skryptu.
> - `logger` bez `-t` wysyła komunikaty z tagiem procesu wywołującego, co utrudnia filtrowanie logów po nazwie skryptu — tag ustawiamy zawsze.

□□ **exit** = zakończ bieżącą powłokę albo skrypt z podanym kodem wyjścia · **sleep** = wstrzymaj wykonywanie skryptu na wskazaną liczbę sekund

Z życia wzięte

Z ŻYCIA WZIĘTE

Niedziela, 23:50. Skrypt backupowy na bkp01, uruchomiony przez systemd, dostaje SIGTERM po przekroczeniu `TimeoutStopSec` w trakcie tworzenia archiwum i kończy się w pół drogi, zostawiając uszkodzony plik `.tar.gz` bez żadnego komunikatu w logu. Dodanie `trap` na `TERM` i `EXIT`, które usuwa niedokończone archiwum i loguje przyczynę przerwania, sprawia, że kolejny taki przypadek zostaje widoczny od razu, zamiast dopiero rano przy próbie przywrócenia danych.

Po co to istnieje

Tryb ścisły, opisany na stronie Tryb ścisły: set -euo pipefail, odpowiada na pytanie „czy się zatrzymać". Ta strona odpowiada na pytania, które zostają otwarte zaraz po zatrzymaniu: co zrobić z plikiem tymczasowym, który skrypt zdążył utworzyć przed awarią; jak zostawić w logu coś więcej niż suchy komunikat basha; jak nie dopuścić, żeby ten sam skrypt backupu uruchomił się drugi raz, zanim pierwszy skończy. `trap` jest mechanizmem, który spina to wszystko w jedno miejsce, uruchamiane niezależnie od tego, w którym miejscu skryptu i z jakiego powodu doszło do zakończenia.

Jak to działa

Rozwiń: jak to działa pod spodem

Pojęcie: sygnał. Komunikat wysyłany przez jądro do procesu, informujący go o zdarzeniu — od żądania zakończenia po informację, że proces potomny zakończył pracę. Proces może zignorować większość sygnałów, obsłużyć je własnym kodem albo pozwolić jądru zastosować domyślne działanie (zwykle zakończenie procesu). Pełną listę i mechanikę numeracji sygnałów opisuje strona Sygnały: kill, SIGTERM, SIGKILL — tutaj interesuje nas, jak powłoka pozwala nam podpiąć własną reakcję.

Pojęcie: trap. Mechanizm powłoki, który rejestruje komendę albo funkcję do wykonania w reakcji na wybrany sygnał albo pseudosygnał. Składnia to `trap 'komenda' SYGNAŁ` albo `trap nazwa_funkcji SYGNAŁ`. Rejestracja przez nazwę funkcji jest czytelniejsza i łatwiejsza w debugowaniu niż wklejanie wielolinijkowego kodu wprost jako napisu do `trap`.

`EXIT` **jest szczególny.** To nie jest sygnał systemowy — to zdarzenie samej powłoki bash, wyzwalane zawsze, gdy skrypt kończy działanie: przez naturalne dojście do ostatniej linii, przez jawny `exit`, przez `set -e` po niepowodzeniu komendy, a nawet po otrzymaniu sygnału, który normalnie kończy proces. Dlatego `trap cleanup EXIT` zarejestrowany raz na początku skryptu gwarantuje sprzątanie w praktycznie każdym scenariuszu zakończenia, bez potrzeby powielania wywołania `cleanup` w każdym miejscu, z którego skrypt może się zakończyć.

`$?` **wewnątrz pułapki** `EXIT`**.** W chwili, gdy powłoka wchodzi do pułapki `EXIT`, zmienna `$?` wciąż niesie kod wyjścia, który spowodował zakończenie — warto odczytać go na samym początku funkcji sprzątającej, zanim jakakolwiek kolejna komenda go nadpisze:

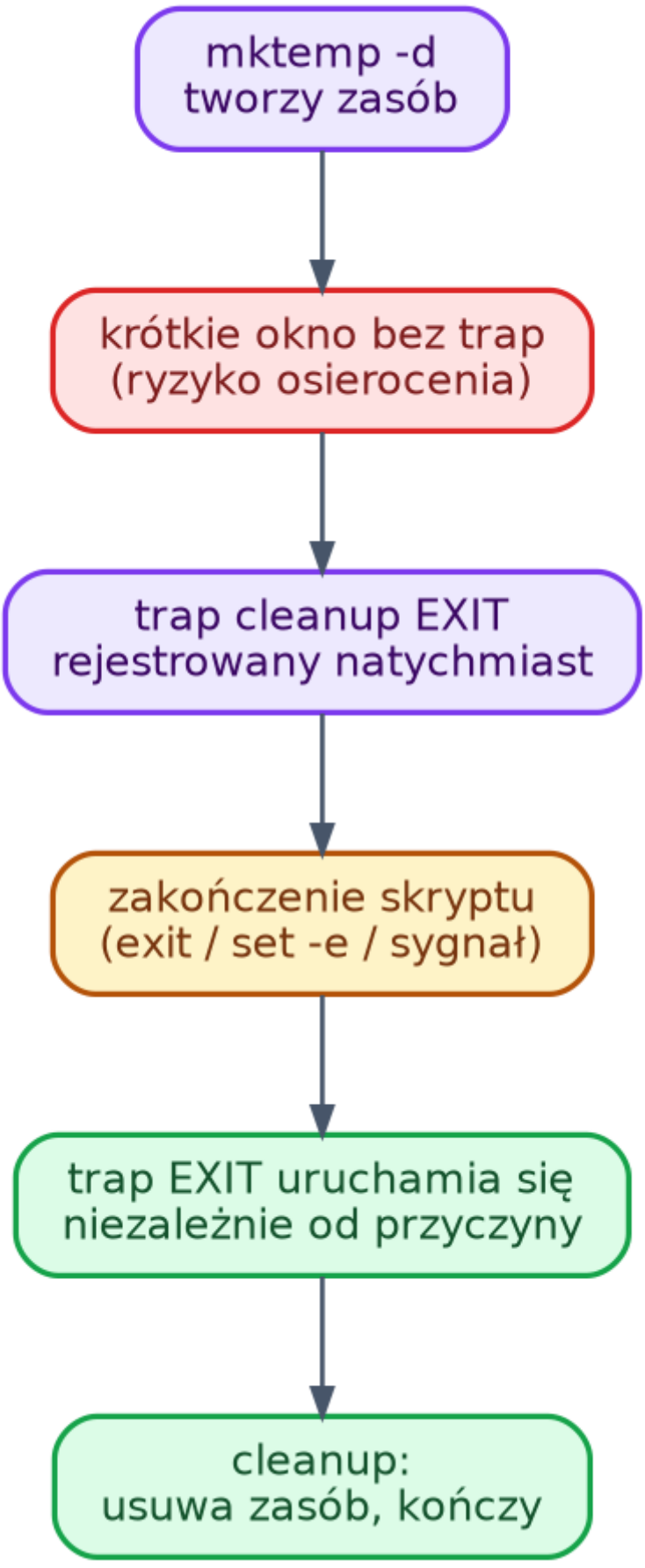
```
cleanup() {
    local kod_wyjscia=$?
    rm -rf -- "$katalog_tymczasowy"
    exit "$kod_wyjscia"
}
trap cleanup EXIT
```

Bez zapisania `kod_wyjscia` na samym początku, `rm` (jeśli się powiedzie) nadpisałby `$?` zerem, i skrypt zgłosiłby sukces nawet po realnym błędzie wcześniej.

`ERR` **i dziedziczenie do funkcji.** Domyślnie pułapka `ERR` zdefiniowana na najwyższym poziomie skryptu nie uruchamia się dla błędu, który wydarzył się wewnątrz funkcji ani wewnątrz podpowłoki — to jest zachowanie zgodne z tym, jak `set -e` samo w sobie traktuje niektóre konteksty, opisane na poprzedniej stronie. `set -E` (`errtrace`) włącza dziedziczenie tej pułapki głębiej, co jest przydatne w skryptach z wieloma funkcjami, gdzie chcemy jednego miejsca do logowania błędu niezależnie od tego, w której funkcji się wydarzył.

Dlaczego `mktemp`**, a nie ręcznie sklejona nazwa.** `plik="/tmp/backup_$$.tmp"` wygląda na unikalne, bo `$$` to PID bieżącego procesu, ale ma dwie realne wady: PID-y są liczbami z ograniczonego zakresu i się powtarzają po zawinięciu licznika, a nazwa w katalogu współdzielonym przez wszystkich użytkowników (`/tmp` ma zwykle bit `sticky`, ale jest światowo zapisywalny) jest przewidywalna z wyprzedzeniem, co przy nieostrożnym kodzie otwiera furtkę na atak przez podmianę pliku, zanim nasz skrypt zdąży go otworzyć (*race condition* znana jako *symlink attack*). `mktemp` tworzy plik atomowo, z losowym komponentem w nazwie, i od razu z bezpiecznymi uprawnieniami ograniczonymi do właściciela.

Kolejność: utworzenie zasobu, zaraz potem `trap`**.** Okno czasowe między `mktemp` a `trap ... EXIT` jest krótkie, ale nie zerowe — jeśli w tym oknie coś przerwie skrypt (na przykład sygnał wysłany dokładnie w tym momencie), plik tymczasowy zostanie osierocony. Zasada praktyczna: rejestrujemy pułapkę sprzątającą w linii bezpośrednio następującej po utworzeniu zasobu, nigdy nie odkładamy tego na później w skrypcie.


mktemp -d
tworzy zasób
krótkie okno bez trap
(ryzyko osierocenia)
trap cleanup EXIT
rejestrowany natychmiast
zakończenie skryptu
(exit / set -e / sygnał)
trap EXIT uruchamia się
niezależnie od przyczyny
cleanup:
usuwa zasób, kończy

Od utworzenia zasobu przez rejestrację `trap` po sprzątanie przy dowolnym zakończeniu skryptu. (kliknięcie otwiera pełny rozmiar)

Krok po kroku

1. Budujemy szkielet z `mktemp`, `trap` i funkcją `die`.

```
#!/bin/bash
set -euo pipefail

die() {
    echo "BŁĄD: $1" >&2
    exit "${2:-1}"
}

katalog_tymczasowy=$(mktemp -d) || die "nie udało się utworzyć katalogu tymczasowego"
cleanup() {
    rm -rf -- "$katalog_tymczasowy"
}
trap cleanup EXIT
```

2. Dodajemy blokadę przed równoległym uruchomieniem.

```
exec 200>/var/run/przyklad.lock
flock -n 200 || die "inna instancja skryptu już działa" 1
```

3. Testujemy zachowanie przy przerwaniu przez Ctrl+C.

```
trap 'echo "przerwano, sprzątam i kończę" >&2; exit 130' INT
sleep 30
```

Uruchamiamy ten fragment i naciskamy `Ctrl+C` w trakcie oczekiwania — pułapka `EXIT` z kroku 1 i tak się wykona, bo `exit 130` też jest zakończeniem skryptu.

Częste błędy

Sprzątanie wpisane bezpośrednio na końcu skryptu, zamiast w `trap`.

```
katalog_tymczasowy=$(mktemp -d)
przetworz_dane "$katalog_tymczasowy"
rm -rf "$katalog_tymczasowy"
```

Jeśli `przetworz_dane` zawiedzie pod `set -e`, skrypt kończy się na tej linii, a `rm -rf` na końcu nigdy się nie wykonuje — katalog tymczasowy zostaje na dysku. Naprawa: `trap` zarejestrowany zaraz po `mktemp -d`, nie sprzątanie dopisane na końcu happy path.

`trap` nadpisany drugim wywołaniem.

```
trap 'echo "zamykam plik A"' EXIT
trap 'echo "zamykam plik B"' EXIT
```

Tylko drugi `trap` przetrwa — pierwszy zostaje bez śladu nadpisany, żadnego ostrzeżenia nie ma. Naprawa: jedna funkcja `cleanup`, która robi obie rzeczy po kolei, zarejestrowana jednym `trap cleanup EXIT`.

Blokada `flock` zwalniana za wcześnie.

```
flock -n /var/run/przyklad.lock -c "dlugie_zadanie.sh"
```

To działa, ale inny, częsty wariant zawodzi:

```
flock -n /var/run/przyklad.lock
dlugie_zadanie.sh
```

Bez `-c` (albo bez otwartego deskryptora przez `exec`) `flock` wywołany jako osobna komenda zwalnia blokadę natychmiast po swoim własnym zakończeniu — `dlugie_zadanie.sh` uruchamia się już bez żadnej ochrony. Poprawny wzorzec trzyma blokadę przez cały czas życia skryptu przez `exec 200>plik` i `flock -n 200`, jak w fiszce wyżej.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — trap EXIT sprząta nawet po błędzie

```
cd /tmp
cat > z_trapem.sh <<'EOF'
#!/bin/bash
set -euo pipefail
tmp=$(mktemp)
trap 'echo "sprzątam: $tmp"; rm -f -- "$tmp"' EXIT
echo "dane robocze" > "$tmp"
false
echo "ta linia się nie wykona"
EOF
chmod +x z_trapem.sh
./z_trapem.sh
echo "kod końcowy: $?"
```

Odpowiedź

```
sprzątam: /tmp/tmp.XXXXXXXXXX
kod końcowy: 1
```

`false` przerywa skrypt pod `set -e`, ale pułapka `EXIT` i tak się uruchamia i usuwa plik tymczasowy, zanim proces faktycznie zakończy działanie. Sprawdzenie `ls /tmp/tmp.*` po uruchomieniu potwierdzi, że plik zniknął.

Ćwiczenie 2 — dwie instancje i flock

```
cd /tmp
cat > z_blokada.sh <<'EOF'
#!/bin/bash
exec 200>/tmp/cw_blokada.lock
flock -n 200 || { echo "zablokowane, wychodzę"; exit 1; }
echo "działam, śpię 5 sekund"
sleep 5
EOF
chmod +x z_blokada.sh
./z_blokada.sh &
sleep 1
./z_blokada.sh
wait
```

Odpowiedź

Druga instancja wypisuje `zablokowane, wychodzę` i kończy się natychmiast z kodem 1, podczas gdy pierwsza wciąż śpi w tle. Blokada trwa przez cały czas życia deskryptora 200, czyli do zakończenia pierwszego procesu.

Pytania kontrolne

1. Dlaczego pułapkę `trap cleanup EXIT` rejestrujemy zaraz po `mktemp`, a nie na samym początku skryptu, przed jego utworzeniem?

Odpowiedź

Pułapka odwołuje się do zmiennej ze ścieżką pliku tymczasowego, która musi już istnieć w chwili jej rejestracji — w praktyce kolejność ma mniejsze znaczenie niż to, żeby między utworzeniem zasobu a rejestracją pułapki nie było żadnej dodatkowej, potencjalnie zawodnej komendy. Zasada brzmi: tworzymy zasób i natychmiast, w kolejnej linii, rejestrujemy jego sprzątanie, żeby żadne przerwanie w międzyczasie nie zostawiło go osieroconym.

2. Skrypt ma `trap cleanup ERR` bez `set -E`. Błąd wydarza się wewnątrz funkcji wywołanej z `main`. Co się stanie?

Odpowiedź

Pułapka `ERR` zarejestrowana na najwyższym poziomie skryptu domyślnie nie dziedziczy się do funkcji, więc `cleanup` może się nie uruchomić dla błędu wewnątrz tej funkcji — zachowanie zależne od kontekstu, w którym błąd wystąpił. Żeby mieć pewność, że `ERR` uruchomi się niezależnie od tego, w której funkcji doszło do błędu, dodajemy `set -E` (`errtrace`) obok `set -euo pipefail`.

Powiązane: Tryb ścisły: set -euo pipefail · Kod wyjścia, $?, &&, ||, średnik · Trzy strumienie: stdin, stdout, stderr · Sygnały: kill, SIGTERM, SIGKILL · Idempotentność: skrypt, który wolno uruchomić dwa razy · Szkielet skryptu produkcyjnego

Źródła: `man 1 bash` (sekcja *SHELL BUILTIN COMMANDS* → `trap`) · `man 1 mktemp` · `man 1 flock` · `man 1 logger` · `man 7 signal` · POSIX.1-2024

Revision #11
Created 2026-09-25 00:40:51 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:22 UTC by Tomasz Konieczny