Kod wyjścia, $?, &&, ||, średnik

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

Każdy proces w Linuksie kończy się liczbą od 0 do 255. Zero znaczy sukces. Na tej jednej liczbie opiera się `&&`, `if`, `set -e`, cron, systemd i każdy skrypt, który ma podjąć decyzję.

Skróty użyte na tej stronie (2)

Skrót	Co znaczy
OOM	Out Of Memory: brak pamięci; OOM killer to mechanizm jądra zabijający proces, gdy jej zabraknie
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Fiszka

ZMIENIA SYSTEM

Kod wyjścia ostatniej komendy

```
grep root /etc/passwd
echo $?
```

Każda zakończona komenda zostawia w powłoce liczbę odczytywalną przez `$?`. Zero oznacza sukces — tu `grep` coś znalazł.

Przykład:

```
root:x:0:0:root:/root:/bin/bash
0
```

grep = global regular expression print: szukaj wzorca w tekście

□□ ZMIENIA SYSTEM

Kod wyjścia przy porażce

```
grep bzzzzt /etc/passwd
echo $?
```

`grep` nic nie znalazł. Nie wypisuje żadnego komunikatu o błędzie — milczy na stdout i zgłasza porażkę wyłącznie liczbą.

Przykład:

```
1
```

□□ **grep** = global regular expression print: szukaj wzorca w tekście

□□ ZMIENIA SYSTEM

Wykonanie po kolei niezależnie od wyniku: `;`

```
mkdir /tmp/x ; cd /tmp/x ; touch plik
```

Średnik tylko sekwencjonuje polecenia. Kod wyjścia poprzedniego nie ma żadnego wpływu na to, czy następne się uruchomi — nawet jeśli `mkdir` zawiedzie, `cd` i tak wystartuje.

□□ **mkdir** = make directory: utwórz katalog

□□ TYLKO ODCZYT

Uruchomienie drugiej komendy tylko po sukcesie: `&&`

```
systemctl is-active --quiet NazwaUslugi && echo "działa"
```

Wykonujemy `echo` wyłącznie wtedy, gdy `systemctl` zwróci kod 0, czyli usługa faktycznie działa.

Przykład:

```
działa
```

□□ **systemctl** = system control: sterowanie usługami systemd

◫◫ ZMIENIA SYSTEM

Uruchomienie drugiej komendy tylko po porażce: `||`

```
ping -c1 -W1 10.20.10.41 >/dev/null || echo "host nie odpowiada"
```

Odwrotność `&&`: prawa strona uruchamia się tylko po porażce lewej.

Przykład:

```
host nie odpowiada
```

◫◫ **ping** = Packet InterNet Groper: sprawdź, czy host odpowiada w sieci i jak długo trwa odpowiedź

◫◫ TYLKO ODCZYT

Negacja kodu wyjścia: `!`

```
! grep -q '^PermitRootLogin no' /etc/ssh/sshd_config && echo "BRAK wpisu"
```

`!` odwraca kod wyjścia następującej po nim komendy: 0 zamienia w 1, a każdy niezerowy kod w 0. Tu: jeśli wpisu nie ma (`grep` zwrócił 1, `!` zmienia to na 0), wypisujemy ostrzeżenie.

Przykład:

```
BRAK wpisu
```

◫◫ TYLKO ODCZYT

Sprawdzenie kodu wyjścia zamiast tekstu: `if`

```
if grep -q 'error' /var/log/messages; then echo "są błędy"; fi
```

`if` nie porównuje żadnych napisów — uruchamia podaną komendę i wchodzi w blok `then` dokładnie wtedy, gdy jej kod wyjścia to 0.

◫◫ TYLKO ODCZYT

Odczyt statusu całego potoku: `PIPESTATUS`

```
false | true
echo $?
echo "${PIPESTATUS[@]}"
```

`$?` po potoku pokazuje kod tylko **ostatniej** komendy. Tablica `PIPESTATUS` trzyma kod każdego członu z osobna, w kolejności — jest ulotna i znika po następnym poleceniu.

Przykład:

```
0
1 0
```

TYLKO ODCZYT

Zwrócenie przez potok ostatniego niezerowego kodu: `set -o pipefail`

```
set -o pipefail
false | true
echo $?
```

Zmienia zasadę odczytu kodu potoku: zamiast kodu ostatniego członu dostajemy pierwszy niezerowy kod z całego łańcucha, a 0 tylko wtedy, gdy wszystkie człony się powiodły.

Przykład:

```
1
```

ZMIENIA SYSTEM

Wypisanie listy sygnałów i ich numerów: `kill -l`

```
kill -l
```

Sygnały mają numery od 1 do 64. Ta lista tłumaczy skąd biorą się kody 128+N: proces zabity sygnałem o numerze N kończy się kodem `128 + N`, więc 130 to `SIGINT`, 137 to `SIGKILL`, a 143 to `SIGTERM`.

Przykład:

```
 1) SIGHUP□ 2) SIGINT□ 3) SIGQUIT□ 4) SIGILL□ 5) SIGTRAP
 6) SIGABRT□ 7) SIGBUS□ 8) SIGFPE□ 9) SIGKILL□10) SIGUSR1
11) SIGSEGV□12) SIGUSR2□13) SIGPIPE□14) SIGALRM□15) SIGTERM
16) SIGSTKFLT□17) SIGCHLD□18) SIGCONT□19) SIGSTOP□20) SIGTSTP
21) SIGTTIN□22) SIGTTOU□23) SIGURG□24) SIGXCPU□25) SIGXFSZ
26) SIGVTALRM□27) SIGPROF□28) SIGWINCH□29) SIGIO□30) SIGPWR
31) SIGSYS□34) SIGRTMIN□35) SIGRTMIN+1□36) SIGRTMIN+2□37) SIGRTMIN+3
38) SIGRTMIN+4□39) SIGRTMIN+5□40) SIGRTMIN+6□41) SIGRTMIN+7□42) SIGRTMIN+8
43) SIGRTMIN+9□44) SIGRTMIN+10□45) SIGRTMIN+11□46) SIGRTMIN+12□47) SIGRTMIN+13
48) SIGRTMIN+14□49) SIGRTMIN+15□50) SIGRTMAX-14□51) SIGRTMAX-13□52) SIGRTMAX-12
53) SIGRTMAX-11□54) SIGRTMAX-10□55) SIGRTMAX-9□56) SIGRTMAX-8□57) SIGRTMAX-7
58) SIGRTMAX-6□59) SIGRTMAX-5□60) SIGRTMAX-4□61) SIGRTMAX-3□62) SIGRTMAX-2
63) SIGRTMAX-1□64) SIGRTMAX
```

□□ **SIGKILL** = sygnał numer 9: każe jądru natychmiast zabić proces, bez szansy na sprzątnięcie po sobie · **SIGTERM** = sygnał numer 15: prośba do procesu, żeby zakończył się grzecznie, proces może ją zignorować albo posprzątać przed wyjściem · **kill** = wyślij sygnał do procesu, domyślnie SIGTERM proszący o zamknięcie

□□ TYLKO ODCZYT

Wyjście ze skryptu z własnym kodem: `exit`

```
exit 2
```

Kończy bieżący skrypt (albo powłokę interaktywną) z podanym kodem. Konwencja: 0 sukces, 1 błąd ogólny, 2 złe użycie.

> **Pojęcie: kod wyjścia.** Liczba od 0 do 255, którą proces oddaje jądru w chwili zakończenia i którą jądro przekazuje procesowi rodzicielskiemu. W powłoce trafia ona do zmiennej `$?`, a operatory `&&`, `||`, `if` i `while` czytają wyłącznie tę liczbę — nigdy tekst, który proces wypisał. Poniższa tabela zbiera konwencje, według których poszczególne wartości są zwyczajowo przyznawane.

□□ **exit** = zakończ bieżącą powłokę albo skrypt z podanym kodem wyjścia

□□ TYLKO ODCZYT

Wszystkie kody

Kod	Znaczenie	Typowa przyczyna
`0`	sukces	—
`1`	błąd ogólny	najczęstszy „coś poszło nie tak"
`2`	błąd użycia / zła składnia	zła flaga, brak argumentu, `ls /nie-ma`
`126`	plik znaleziony, ale nie da się wykonać	brak bitu `x`, katalog zamiast programu
`127`	komenda nie znaleziona	literówka, brak w `$PATH`, zły shebang
`128+N`	zabity sygnałem N	patrz `kill -l` wyżej
`130`	`SIGINT` (2)	ktoś nacisnął `Ctrl+C`
`137`	`SIGKILL` (9)	`kill -9` albo OOM killer
`143`	`SIGTERM` (15)	`systemctl stop`, `kill` bez flag

Uważaj

- `$?` odnosi się do **ostatniej wykonanej komendy** — nawet do `echo`. Sprawdzaj go natychmiast albo zapisz do zmiennej.
- `a && b || c` nie jest odpowiednikiem `if a then b else c` — `c` wykona się także wtedy, gdy `b` zawiedzie.
- `exit 256` daje kod **0**, bo liczby są obcinane modulo 256. `exit -1` daje 255.

OOM = Out Of Memory: brak pamięci; OOM killer to mechanizm jądra zabijający proces, gdy jej zabraknie

Z życia wzięte

Z ŻYCIA WZIĘTE

Piątek, 16:00, więc oczywiście coś przestaje działać. Usługa na `app01` pada z kodem 137 i Tomek godzinę szuka błędu w konfiguracji, zanim ktoś mu przypomina wzór `128 + numer sygnału`. 137 minus 128 daje 9, czyli `SIGKILL`: `dmesg -T | grep -i 'killed process'` potwierdza, że to OOM killer ubił proces za zużycie pamięci, a nie błąd w kodzie aplikacji.

Po co to istnieje

Piszesz skrypt, który robi kopię bazy i wysyła ją na `bkp01.zielona.example`. Stara kopia ma zostać usunięta dopiero wtedy, gdy nowa faktycznie doleciała. Skąd skrypt ma wiedzieć, czy `rsync` się udał? Nie może czytać komunikatów po polsku i zgadywać — potrzebuje sygnału maszynowego, jednoznacznego, tego samego dla każdego programu na świecie.

Tym sygnałem jest **kod wyjścia** (*exit status*): jedna liczba, którą proces oddaje jądru w chwili śmierci, a jądro przekazuje procesowi rodzicielskiemu. Powłoka podstawia ją do zmiennej `$?` — `rsync -a /var/backup/ jan.nowak@bkp01.zielona.example:/kopie/` i zaraz potem `echo $?` wypisze `0`, jeśli transfer się powiódł.

Wszystko, co robi powłoka w zakresie podejmowania decyzji — `&&`, `||`, `if`, `while`, `set -e` — sprowadza się do porównania tej liczby z zerem. Cron wysyła maila, gdy zadanie zwróci coś innego niż 0. systemd restartuje usługę, gdy proces skończy się niezerowo.

Analogia

Wyobraź sobie **protokół kontroli jakości na końcu linii produkcyjnej**.

Kontroler bierze wyrób, sprawdza go i wypełnia jedną rubrykę na karcie: numer. Wpisuje **0**, gdy nie ma uwag. Jeśli coś jest nie tak, wpisuje numer usterki z zakładowego katalogu: 1 to „wada ogólna", 2 to „zły dokument towarzyszący", 126 to „nie dało się uruchomić urządzenia testowego", 127 to „nie znaleziono stanowiska kontrolnego".

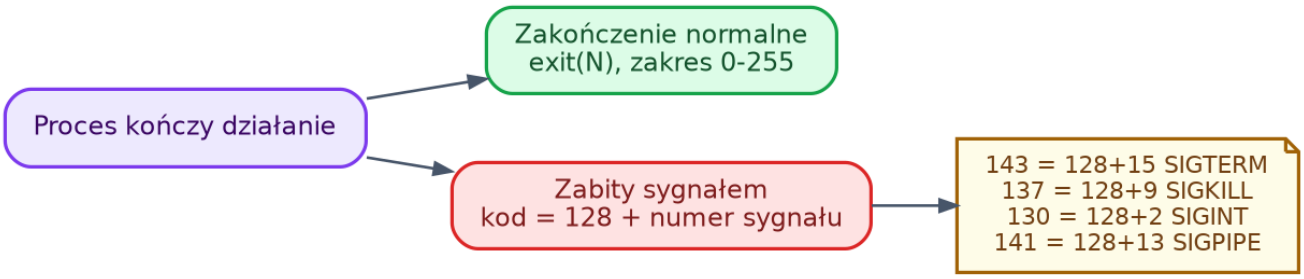
Ta konwencja jest odwrotna do intuicji i właśnie dlatego jest praktyczna. Sposobów na sukces jest dokładnie jeden, więc wystarczy mu jeden numer — zero. Sposobów na porażkę są dziesiątki i każdy zasługuje na własny kod, bo kto inaczej zareaguje na „brak dokumentu", a inaczej na „urządzenie się nie włącza".

Analogia niesie dwa ograniczenia, które tłumaczą realne pułapki. **Rubryka jest jedna i przy następnym wyrobie zostaje nadpisana** — jeśli nie przepiszesz numeru do rejestru od razu, przepadnie. Dokładnie tak działa `$?`, który po każdej kolejnej komendzie, także po niewinnym `echo`, wskazuje już na nią. I drugie: **katalog usterek ma numery od 0 do 255**, a kontroler nie może wpisać nic spoza listy — próba wpisania 300 kończy się obcięciem do 44. To nie jest teoria: `exit 256` w skrypcie raportuje sukces po błędzie.

Jak to działa

Rozwiń: jak to działa pod spodem

Gdy proces kończy pracę, wywołuje `exit(N)` — a właściwie systemowe `_exit()`. Jądro zapamiętuje ten status i zostawia proces w stanie **zombie**, dopóki rodzic nie odbierze wyniku przez `wait()`. Dopiero wtedy wpis w tabeli procesów znika.



Dwie drogi kodu wyjścia: zakończenie normalne 0-255 albo 128 plus numer sygnału (kliknięcie otwiera pełny rozmiar)

Ta liczba nie jest zwykłym intem — jądro pakuje do niej kilka informacji naraz. Przy normalnym zakończeniu w statusie siedzi 8 bitów kodu, czyli zakres **0–255**. Gdy proces został **zabity sygnałem**, status zawiera numer sygnału, a powłoka raportuje go jako `128 + numer`.

Ten mechanizm tłumaczy kody, które na pierwszy rzut oka wyglądają na przypadkowe. Proces zatrzymany przez `systemctl stop` kończy się kodem 143, a nie 15, bo 143 = 128 + 15, a 15 to numer sygnału `SIGTERM`. Analogicznie 137 = 128 + 9 (`SIGKILL`, czyli `kill -9` albo OOM killer), 130 = 128 + 2 (`SIGINT`, czyli `Ctrl+C`), 141 = 128 + 13 (`SIGPIPE`, czyli ktoś zamknął drugi koniec potoku). Pełną listę numerów sygnałów pokazuje `kill -l`.

Ten podział ma praktyczne znaczenie przy diagnozie. Jeśli usługa kończy się z kodem 137, nie ma sensu szukać błędu w jej konfiguracji — ona nie zdążyła niczego zdecydować. Ktoś ją zabił, najprawdopodobniej OOM killer; sprawdzasz `dmesg -T | grep -i 'killed process'` i masz odpowiedź.

`$?` **a operatory.** Powłoka podstawia do `$?` kod ostatnio zakończonego polecenia pierwszego planu. Operatory logiczne po prostu ten kod czytają:

- `A ; B` — wykonaj `A`, potem `B`. Kod `A` nie ma znaczenia.
- `A && B` — wykonaj `B` tylko wtedy, gdy `A` zwróciło 0.
- `A || B` — wykonaj `B` tylko wtedy, gdy `A` zwróciło coś innego niż 0.

Oba operatory są **leniwe**: jeśli `A && B` przerwie na `A`, `B` nigdy nie wystartuje, a kodem całości jest kod `A`. Wiążą też lewostronnie z równym priorytetem, co jest źródłem częstego błędu opisanego niżej, w sekcji „Częste błędy".

`if` **nie testuje tekstu.** To jest moment, w którym ludziom przychodzącym z innych języków rozjeżdża się model:

```
if grep -q 'PermitRootLogin no' /etc/ssh/sshd_config; then
    echo "root nie może się logować"
fi
```

Nie ma tu żadnego porównania. `if` uruchamia `grep`, czeka na jego kod wyjścia i wchodzi do bloku, gdy kod to 0. `grep -q` zwraca 0, gdy coś znalazł, i 1, gdy nie znalazł.

Z tego wynika rzecz, która zaskakuje: `[` **to nie składnia, tylko program.**

Sprawdź `ls -l /usr/bin/[` — naprawdę istnieje plik wykonywalny o tej nazwie. W bashu używana jest wbudowana wersja, ale zasada pozostaje: `[ -f /etc/passwd ]` to wywołanie programu z argumentami `-f`, `/etc/passwd` i `]`, który zwraca 0 albo 1. Dlatego spacje wokół nawiasów są obowiązkowe — bez nich powłoka szuka komendy o nazwie `[-f`.

Krok po kroku

1. Sprawdź kod wyjścia i zapisz go.

```
systemctl is-active --quiet nginx
kod=$?
echo "kod=$kod"
```

Przypisanie do zmiennej robisz **natychmiast**. Każda komenda między nimi — nawet `echo` — nadpisze `$?`.

2. Łączenie warunkowe.

```
mkdir -p /tmp/praca && cd /tmp/praca && touch raport.txt
```

Jeśli `mkdir` zawiedzie, `cd` się nie wykona, a `touch` tym bardziej. Bez `&&` skrypt utworzyłby `raport.txt` tam, gdzie akurat go uruchomiono.

3. Standardowe kody w praktyce.

```
nieistniejaca_komenda ; echo "kod=$?"
```

```
bash: nieistniejaca_komenda: command not found
kod=127
```

```
touch /tmp/skrypt.sh ; chmod 644 /tmp/skrypt.sh ; /tmp/skrypt.sh ; echo "kod=$?"
```

```
bash: /tmp/skrypt.sh: Permission denied
kod=126
```

Zapamiętaj różnicę: **127 = nie znalazłem, 126 = znalazłem, ale nie mogę uruchomić**. To rozróżnienie oszczędza sporo czasu przy debugowaniu crona, gdzie `$PATH` jest okrojony i 127 pojawia się nagminnie.

4. Kody wyjścia w skrypcie.

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

[ $# -eq 2 ] || { echo "Użycie: $0 ZRODLO CEL" >&2; exit 2; }
[ -d "$1" ] || { echo "Katalog $1 nie istnieje" >&2; exit 1; }

rsync -a "$1" "$2" || exit 3
exit 0
```

Komunikaty błędów idą na stderr (`>&2`), żeby ktoś, kto puści Twój skrypt w potoku, dostał czyste dane. Skrypt bez jawnego `exit` kończy się kodem ostatniej wykonanej komendy — jeśli ostatnią linią jest `echo "gotowe"`, zwróci 0 niezależnie od tego, co się wcześniej wydarzyło.

5. Kod wyjścia potoku.

```
false | true ; echo "kod potoku=$?"
echo "${PIPESTATUS[@]}"
```

```
kod potoku=0
1 0
```

> **Pojęcie: status potoku (`PIPESTATUS`).** Zmienna `$?` po potoku niesie tylko kod **ostatniego** członu — tak jest zdefiniowane w POSIX i tak działa `false | true` powyżej: kod całości to 0, mimo że `false` zawiodło. Tablica `PIPESTATUS` (rozszerzenie basha) trzyma kod każdego programu z osobna, w kolejności, w jakiej stały w potoku — `${PIPESTATUS[0]}` to kod pierwszego członu, `${PIPESTATUS[1]}` drugiego i tak dalej. Jest ulotna: kolejna komenda ją nadpisuje, więc trzeba ją skopiować od razu po potoku.

`set -o pipefail` zmienia zasadę: potok zwraca ostatni niezerowy kod z całego łańcucha, a 0 tylko wtedy, gdy wszystkie człony się powiodły.

6. Kody wyjścia w systemd.

Plik jednostki z `Restart=on-failure` restartuje usługę tylko wtedy, gdy proces skończył się **niezerowo** albo został zabity sygnałem:

```
[Service]
ExecStart=/usr/local/bin/agent.sh
Restart=on-failure
RestartSec=5
```

Jeśli skrypt skończy się `exit 0`, `systemctl status agent` pokaże `Active: inactive (dead)` i `Process: ... (code=exited, status=0/SUCCESS)`.

Usługa nie wstała, bo `status=0` to dla systemd sukces: proces wykonał zadanie i grzecznie zakończył pracę, nie ma czego restartować. Jeśli chcesz restartu zawsze — `Restart=always`; jeśli konkretny niezerowy kod ma być uznany za sukces — `SuccessExitStatus=3`. Kody 200–242 (`203/EXEC` gdy pliku nie da się uruchomić, `200/CHDIR` przy złym `WorkingDirectory`) pochodzą z wewnętrznej puli systemd, nie z Twojego programu — lista w `systemd.exec(5)`.

Częste błędy

Sprawdzenie `$?` o jedną komendę za późno.

```
rsync -a /var/backup/ jan.nowak@bkp01.zielona.example:/kopie/
echo "rsync zakończony"
if [ $? -ne 0 ]; then
    echo "BŁĄD transferu" >&2
    exit 1
fi
```

Ten kod **nigdy** nie zgłosi błędu. `$?` w warunku to kod wyjścia `echo`, a `echo` prawie zawsze kończy się zerem. Skrypt raportuje sukces po każdej awarii transferu.

Naprawa — zapisz kod natychmiast po komendzie (`kod=$?`, potem `if [ "$kod" -ne 0 ]`) albo, lepiej, w ogóle nie używaj `$?`:

```
if ! rsync -a /var/backup/ jan.nowak@bkp01.zielona.example:/kopie/; then
    echo "BŁĄD transferu" >&2
    exit 1
```

```
fi
```

Druga wersja jest odporna na przypadkowe wstawienie linii w środek.

`a && b || c` potraktowane jak if-then-else.

```
[ -f /etc/myapp.conf ] && cp /etc/myapp.conf /backup/ || echo "brak pliku konfiguracyjnego"
```

Wygląda niewinnie. Ale jeśli plik **istnieje**, a `cp` zawiedzie (brak miejsca na `/backup`, brak uprawnień, katalog tylko do odczytu), to `||` zobaczy niezerowy kod od `cp` i wypisze „brak pliku konfiguracyjnego". Komunikat jest fałszywy i wysyła Cię na godzinę w złą stronę.

Sprawdzisz to jedną linią:

```
true && false || echo "c się wykonało"
```

```
c się wykonało
```

Konstrukcja `a && b || c` jest bezpieczna tylko wtedy, gdy `b` nie może zawieść (np. `echo`). W każdym innym przypadku pisz pełne `if`:

```
if [ -f /etc/myapp.conf ]; then
    cp /etc/myapp.conf /backup/
else
    echo "brak pliku konfiguracyjnego" >&2
fi
```

`if [ $? ]` zamiast `if [ $? -eq 0 ]`.

```
grep -q wzorzec /etc/hostname
if [ $? ]; then echo "znaleziono"; fi
```

```
znaleziono
```

Wypisze „znaleziono" zawsze, także gdy `grep` nic nie znalazł. `[ napis ]` z jednym argumentem testuje, czy **napis jest niepusty** — a `1` jest napisem niepustym, więc test przechodzi. Potrzebne jest jawne porównanie: `[ $? -eq 0 ]`. Albo, prościej, `if grep -q ...; then`.

`exit` z liczbą spoza zakresu.

```
bash -c 'exit 256' ; echo "kod=$?"
```

```
kod=0
```

Skrypt, który chciał zgłosić błąd numer 256, zaraportował pełny sukces — status jest obcinany do 8 bitów, a 256 mod 256 = 0. Podobnie `exit -1` daje 255, a `exit 300` daje 44. Trzymaj własne kody w zakresie 1–125; powyżej zaczyna się terytorium powłoki (126, 127) i sygnałów (128+).

□□ Ćwiczenia

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

Ćwiczenie 1 — katalog kodów

```
cd /tmp
true ; echo "true -> $?"
false ; echo "false -> $?"
ls /nie-ma-takiego-katalogu 2>/dev/null ; echo "ls -> $?"
nieistniejaca_komenda 2>/dev/null ; echo "brak komendy -> $?"
touch nowy.sh ; chmod 644 nowy.sh ; ./nowy.sh 2>/dev/null ; echo "bez x -> $?"
sleep 60 & kill -9 $! ; wait $! 2>/dev/null ; echo "kill -9 -> $?"
```

Odpowiedź

```
true -> 0
false -> 1
ls -> 2
brak komendy -> 127
bez x -> 126
kill -9 -> 137
```

`ls` używa 2 dla „nie ma takiego pliku", zgodnie z konwencją GNU (2 = poważny problem, 1 = drobny). 137 to 128 + 9, czyli `SIGKILL`.

Ćwiczenie 2 — pułapka `&& ||`

```
cd /tmp
printf 'dane\n' > zrodlo.txt
```

```
: > cel                      # "cel" jest zwykłym plikiem, nie katalogiem

[ -f zrodlo.txt ] && cp zrodlo.txt cel/ || echo "brak pliku źródłowego"
```

Plik `zrodlo.txt` istnieje. Co zobaczysz i dlaczego?

Odpowiedź

```
cp: cannot stat 'cel/': Not a directory
brak pliku źródłowego
```

Komunikat jest kłamstwem — plik źródłowy istnieje i test go znalazł. `||` zareagował na niezerowy kod `cp`, nie na wynik `[ -f ]`. To dokładnie ten scenariusz, który w skrypcie wysyła Cię na godzinę w złą stronę. Popraw przez pełne `if/else`, posprzątaj: `rm /tmp/zrodlo.txt /tmp/cel`.

Ćwiczenie 3 — własny skrypt z kodami

Napisz w `/tmp/sprawdz.sh` skrypt, który dostaje ścieżkę jako argument i zwraca: 2 gdy brak argumentu, 1 gdy plik nie istnieje, 0 gdy istnieje.

Odpowiedź

```
cat > /tmp/sprawdz.sh <<'EOF'
#!/bin/bash
[ $# -eq 1 ] || { echo "Użycie: $0 SCIEZKA" >&2; exit 2; }
[ -e "$1" ] || { echo "$1 nie istnieje" >&2; exit 1; }
exit 0
EOF
chmod +x /tmp/sprawdz.sh

/tmp/sprawdz.sh            ; echo "brak arg -> $?"
/tmp/sprawdz.sh /nie-ma    ; echo "brak pliku -> $?"
/tmp/sprawdz.sh /etc/hosts ; echo "jest -> $?"
```

```
brak arg -> 2
brak pliku -> 1
```

```
jest -> 0
```

Pytania kontrolne

1. Usługa ma `Restart=on-failure`. Skrypt kończy się `exit 0` po napotkaniu błędu, bo jego autor uznał, że „przecież sam obsłużył sytuację". Co zrobi systemd?

Odpowiedź

Nic. `Restart=on-failure` reaguje wyłącznie na niezerowy kod wyjścia albo śmierć od sygnału. `exit 0` to dla systemd komunikat „zadanie wykonane poprawnie". Usługa przejdzie w stan `inactive (dead)` i zostanie tak. Naprawa: skrypt musi kończyć się niezerowo przy błędzie, albo jednostka potrzebuje `Restart=always`.

2. `cmd1 && cmd2 || cmd3` — w ilu sytuacjach wykona się `cmd3`?

Odpowiedź

W dwóch: gdy `cmd1` zwróci niezerowy kod (wtedy `cmd2` w ogóle nie startuje) oraz gdy `cmd1` się powiedzie, ale niezerowy kod zwróci `cmd2`. `||` patrzy na kod ostatniej faktycznie wykonanej komendy z lewej strony. Bezpieczne tylko wtedy, gdy `cmd2` nie może zawieść.

3. Proces kończy się z kodem 137. Co się stało i czego NIE ma sensu sprawdzać?

Odpowiedź

137 = 128 + 9, czyli proces został zabity sygnałem `SIGKILL`. Nie zakończył się sam, więc nie ma sensu szukać przyczyny w jego konfiguracji ani w logach aplikacji — nie zdążyła zareagować, bo `SIGKILL` jest nieprzechwytywalny. Najczęstsze źródła: OOM killer (`dmesg -T | grep -i 'killed process'`), ręczne `kill -9`, albo `TimeoutStopSec` w systemd, po którym usługa dostaje `SIGKILL`.

4. Dlaczego `if [ -f /etc/passwd ]` wymaga spacji wokół nawiasów, a `if (x > 0)` w C nie?

Odpowiedź

Bo `[` nie jest w powłoce elementem składni, tylko nazwą komendy — istnieje nawet plik `/usr/bin/[`. Powłoka dzieli linię na słowa po spacjach, więc `[-f` byłaby nazwą komendy, której

nie ma. `[` przyjmuje argumenty (`-f`, ścieżkę, zamykający `]`) i zwraca kod wyjścia 0 albo 1, a `if` na ten kod reaguje. To zwykłe wywołanie programu, nie wyrażenie logiczne.

Powiązane: Potoki i filozofia Unix · Trzy strumienie: stdin, stdout, stderr · Sygnały: kill, SIGTERM, SIGKILL · set -euo pipefail i pisanie bezpiecznych skryptów · Jednostki systemd: pliki .service · Zadania cron i ich diagnostyka

Źródła: `man 1 bash` (sekcje *Exit Status*, *Lists*, *Pipelines*) · `man 1 exit` · `man 1 test` · `man 1 kill` · `man 5 systemd.exec` (sekcja *Process Exit Codes*) · `man 7 signal` · POSIX.1-2024

Revision #19
Created 2026-09-17 11:56:19 UTC by Tomasz Konieczny
Updated 2026-10-03 19:07:29 UTC by Tomasz Konieczny