Sygnały: kill, pkill, killall, trap

Poziom: średnio **Dotyczy: wszystkie dystrybucje** **Czas: ~16 min**

Sygnał to przerwanie, które jądro dostarcza procesowi — od grzecznej prośby o zakończenie po natychmiastowe ścięcie bez ostrzeżenia. `kill` swoją nazwą sugeruje jedno działanie, a wysyła dowolny sygnał z całej puli.

Skróty użyte na tej stronie (7)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
B2B	business to business: sprzedaż i systemy skierowane do innych firm, nie do klientów indywidualnych
NFS	Network File System: udostępnianie katalogów po sieci w świecie Linuksa
PID	Process ID: numer procesu w systemie
RHEL	Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa
SLES	SUSE Linux Enterprise Server: płatna dystrybucja Linuksa firmy SUSE, konkurent Rocky/RHEL
SSH	Secure Shell: szyfrowane zdalne logowanie do serwera

Fiszka

Pojęcie: sygnał. Jednobajtowy komunikat, który jądro dostarcza procesowi w dowolnym momencie jego działania, przerywając jego normalny bieg. Nie niesie żadnych danych poza swoim numerem — to informacja „zdarzenie numer N", nic więcej. Wysłać go może inny proces (przez `kill`), samo jądro (np. przy dzieleniu przez zero albo rozłączeniu terminala), albo my sami z klawiatury (`Ctrl+C`, `Ctrl+Z`). Poniższa lista wylicza wszystkie sygnały dostępne w systemie.

ZMIENIA SYSTEM

Wypisanie listy sygnałów: `kill -l`

```
kill -l
```

Wypisujemy nazwy i numery wszystkich sygnałów zdefiniowanych w systemie. Sięgamy po to, gdy nie pamiętamy, jaki numer ma `SIGUSR1` albo `SIGHUP`.

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
```

SIGHUP = sygnał wysyłany, gdy zamykamy terminal, w którym proces wystartował, dawniej oznaczał rozłączenie linii telefonicznej · **kill** = wyślij sygnał do procesu, domyślnie SIGTERM proszący o zamknięcie

ZMIENIA SYSTEM

Zakończenie procesu domyślnym sygnałem: `kill`

```
kill NumerPID
```

Bez żadnej flagi `kill` wysyła `SIGTERM` (numer 15) — grzeczną prośbę o zakończenie. To jest domyślne zachowanie tej komendy, mimo że sama nazwa sugeruje coś bardziej brutalnego.

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

ZMIENIA SYSTEM

Jawne żądanie zakończenia procesu: `kill -TERM`

```
kill -TERM NumerPID
```

To samo co `kill NumerPID` bez flagi, tylko zapisane wprost. Warto pisać `-TERM` jawnie, gdy komenda trafia do skryptu, który ktoś inny będzie czytał później i nie powinien się domyślać, co robi goły `kill`.

kill = wyślij sygnał do procesu, domyślnie SIGTERM proszący o zamknięcie

□□ OSTROŻNIE · NIEODWRACALNE

Natychmiastowe zabicie procesu: `kill -9` / `kill -KILL`

```
kill -9 NumerPID
```

Wysyłamy `SIGKILL` (numer 9) — proces znika natychmiast, bez żadnej szansy na posprzątanie. Ostateczność, nie pierwszy odruch: patrz *Częste błędy*.

□□ **SIGKILL** = sygnał numer 9: każe jądru natychmiast zabić proces, bez szansy na sprzątnięcie po sobie · **kill** = wyślij sygnał do procesu, domyślnie SIGTERM proszący o zamknięcie

□□ ZMIENIA SYSTEM

Przeładowanie konfiguracji demona: `kill -HUP`

```
kill -HUP NumerPID
```

U większości demonów systemowych umowny sygnał „przeładuj plik konfiguracyjny, ale nie kończ procesu". Nie każda usługa to obsługuje — trzeba sprawdzić dokumentację konkretnej aplikacji.

□□ **kill** = wyślij sygnał do procesu, domyślnie SIGTERM proszący o zamknięcie

□□ ZMIENIA SYSTEM

Sprawdzenie, czy proces istnieje: `kill -0`

```
kill -0 NumerPID && echo "żyje" || echo "nie ma takiego procesu"
```

Nie wysyła żadnego realnego sygnału — jądro tylko sprawdza uprawnienia i istnienie PID-u. Przydaje się w skryptach, które chcą wiedzieć, czy proces z zapisanego wcześniej pliku `.pid` wciąż działa.

Przykład:

```
żyje
```

□□ **kill** = wyślij sygnał do procesu, domyślnie SIGTERM proszący o zamknięcie

□□ ZMIENIA SYSTEM

Wysłanie sygnału do zadania powłoki: `kill %1`

```
kill %1
```

Zamiast PID-u można podać numer zadania bieżącej powłoki (`%1`, `%2`…) — działa z każdą flagą, np. `kill -TERM %1`. Pełne omówienie numerów zadań jest na stronie *Zadania powłoki: jobs, bg, fg, nohup, disown*.

kill = wyślij sygnał do procesu, domyślnie SIGTERM proszący o zamknięcie

TYLKO ODCZYT

Szukanie PID-ów po nazwie: `pgrep`

```
pgrep nginx
```

Wypisujemy same numery PID procesów, których nazwa albo linia poleceń pasuje do wzorca — bez ręcznego przeszukiwania wyniku `ps`.

Przykład:

```
1421
1487
1488
```

PID = Process ID: numer procesu w systemie · **pgrep** = process grep: znajdź numer procesu po nazwie bez przeglądania pełnej listy ps

TYLKO ODCZYT

Wypisanie pełnej linii poleceń: `pgrep -af`

```
pgrep -af nginx
```

Dokłada do każdego PID-u całą linię poleceń, którą proces został uruchomiony. Zawsze uruchamiamy to najpierw, zanim wyślemy cokolwiek przez `pkill -f` — żeby zobaczyć, co dokładnie złapiemy.

Przykład:

```
1421 nginx: master process /usr/sbin/nginx
1487 nginx: worker process
```

```
1488 nginx: worker process
```

pgrep = process grep: znajdź numer procesu po nazwie bez przeglądania pełnej listy ps

TYLKO ODCZYT

Filtrowanie po właścicielu: `pgrep -u`

```
pgrep -u nazwa_usera -a python3
```

Zawężamy dopasowanie do procesów należących do konkretnego użytkownika — przydatne na serwerze, gdzie kilka osób uruchamia procesy o podobnych nazwach.

pgrep = process grep: znajdź numer procesu po nazwie bez przeglądania pełnej listy ps

TYLKO ODCZYT

Zabicie procesu po nazwie: `pkill`

```
pkill worker
```

Łączymy `pgrep` z `kill` w jednym kroku: znajdujemy pasujące procesy i od razu wysyłamy im sygnał (domyślnie `SIGTERM`), bez pośredniego etapu z PID-em.

SIGTERM = sygnał numer 15: prośba do procesu, żeby zakończył się grzecznie, proces może ją zignorować albo posprzątać przed wyjściem · **pkill** = process kill: wysyła sygnał do procesów wybranych po nazwie; z opcją -9 kończy je natychmiast, co w testach udaje awarię

TYLKO ODCZYT

Dopasowanie do całej linii poleceń: `pkill -f`

```
pkill -TERM -f 'worker.py --queue=default'
```

Dopasowuje wzorzec do **całej linii poleceń**, nie tylko do nazwy programu — dlatego zawężamy go precyzyjnie, a nie do samej nazwy skryptu. Ryzyko szerokiego trafienia opisane w *Częste błędy*.

pkill = process kill: wysyła sygnał do procesów wybranych po nazwie; z opcją -9 kończy je natychmiast, co w testach udaje awarię

TYLKO ODCZYT

Zabicie procesów danego użytkownika: `pkill -u`

```
pkill -u nazwa_usera -x sleep
```

`-u` ogranicza dopasowanie do procesów należących do wskazanego użytkownika, `-x` wymaga dokładnego dopasowania nazwy zamiast dowolnego fragmentu.

pkill = process kill: wysyła sygnał do procesów wybranych po nazwie; z opcją -9 kończy je natychmiast, co w testach udaje awarię

TYLKO ODCZYT

Zabicie procesu po dokładnej nazwie polecenia: `killall`

```
killall -TERM nginx
```

W odróżnieniu od `pkill -f`, `killall` (pakiet `psmisc`, ten sam na RHEL/Rocky, Debianie/Ubuntu i SLES) dopasowuje do **nazwy polecenia** (`comm`), nie do dowolnego fragmentu linii poleceń. Dodanie `-u nazwa_usera` zawęża go do procesów jednego użytkownika — ważne na serwerze współdzielonym, patrz *Częste błędy*.

RHEL = Red Hat Enterprise Linux: komercyjna dystrybucja Linuksa · **SLES** = SUSE Linux Enterprise Server: płatna dystrybucja Linuksa firmy SUSE, konkurent Rocky/RHEL

TYLKO ODCZYT

Ustawienie limitu czasu na wykonanie komendy: `timeout`

```
timeout -k 5 30 dlugi_skrypt.sh
```

Uruchamiamy komendę, a po 30 sekundach `timeout` wysyła jej `SIGTERM`; jeśli proces nie zakończy się w kolejne 5 sekund (`-k`), dobija go `SIGKILL`. Zabezpieczenie przed zawieszonym poleceniem w skrypcie albo w cronie. Kod wyjścia `124` oznacza przekroczenie limitu.

SIGTERM = sygnał numer 15: prośba do procesu, żeby zakończył się grzecznie, proces może ją zignorować albo posprzątać przed wyjściem · **SIGKILL** = sygnał numer 9: każe jądru natychmiast zabić proces, bez szansy na sprzątnięcie po sobie · **timeout** = uruchom polecenie i przerwij je, jeśli nie skończy się w zadanym czasie

◫◫ OSTROŻNIE · NIEODWRACALNE

Sprzątanie przy wyjściu ze skryptu: `trap ... EXIT`

```
trap 'rm -f /tmp/moj.lock; echo "posprzątane"' EXIT
```

Kod w cudzysłowie wykonuje się zawsze, gdy skrypt kończy działanie — niezależnie od tego, czy skończył się normalnie, przez `exit`, czy przez przerwanie z klawiatury.

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

◫◫ TYLKO ODCZYT

Przechwycenie przerwania: `trap ... INT TERM`

```
trap 'echo "przerwano, kończę"; exit 1' INT TERM
```

To samo co wyżej, ale dla konkretnych sygnałów zamiast samego wyjścia — pozwala zareagować inaczej na `Ctrl+C` (`SIGINT`) niż na zwykłe zakończenie skryptu. Przykład działania przy realnym przerwaniu jest w *Krok po kroku*.

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

◫◫ TYLKO ODCZYT

Wszystkie flagi

Komenda / flaga	Co robi	Kiedy używasz
`kill -l`	wypisuje nazwy i numery wszystkich sygnałów	nie pamiętasz, jaki numer ma `SIGUSR1`
`kill -TERM PID`	prosi proces o zakończenie (domyślne dla `kill`)	pierwszy odruch przy zamykaniu czegokolwiek
`kill -9 PID` / `kill -KILL PID`	ubija proces natychmiast, bez szansy na sprzątanie	proces nie reaguje na `TERM` po odczekaniu
`kill -0 PID`	nic nie wysyła, tylko sprawdza istnienie procesu	skrypt chce wiedzieć, czy PID z pliku `.pid` wciąż żyje
`kill -HUP PID`	umowny sygnał „przeładuj konfigurację" u demonów	zmieniłeś config i nie chcesz restartować usługi
`pgrep -af wzorzec`	pokazuje PID i pełną linię poleceń pasujących procesów	zawsze PRZED użyciem `pkill -f`

Komenda / flaga	Co robi	Kiedy używasz
`pkill -f wzorzec`	zabija wszystkie procesy pasujące do wzorca w linii poleceń	wygodne, ale niebezpieczne — patrz Częste błędy
`killall nazwa`	zabija po dokładnej nazwie polecenia (`comm`)	znasz dokładną nazwę binarki
`timeout N komenda`	ubija komendę, jeśli działa dłużej niż N sekund	zabezpieczenie przed zawieszonym poleceniem w skrypcie/cronie
`trap 'kod' SYGNAŁ`	wykonuje kod, gdy skrypt dostanie dany sygnał	sprzątanie plików tymczasowych przy `Ctrl+C`

Uważaj

- `SIGKILL` (9) i `SIGSTOP` (19) **nie da się przechwycić ani zignorować** — proces nie ma szansy nic zrobić przed zakończeniem.
- `pkill -f wzorzec` dopasowuje do **całej linii poleceń**, nie tylko nazwy procesu — łatwo trafić w coś, czego nie chciałeś.
- Kod wyjścia procesu zabitego sygnałem to `128 + numer sygnału` (np. `137` po `SIGKILL`) — pełny opis w *Kod wyjścia, $?, &&, ||, średnik*.

PID = Process ID: numer procesu w systemie · **SIGKILL** = sygnał numer 9: każe jądru natychmiast zabić proces, bez szansy na sprzątnięcie po sobie

Z życia wzięte

Z ŻYCIA WZIĘTE

W piątek o 16:00 Ola prosi o przeładowanie konfiguracji nginksa na app01 bez zrywania trwających połączeń klientów sklepu B2B. Tomek sięga po `kill -HUP` zamiast `systemctl restart`, bo restart zamknąłby wszystkie gniazda naraz. Proces wczytuje nowy config i pracuje dalej pod tym samym PID-em, a nikt z klientów nawet nie zauważa przerwy.

Po co to istnieje

Proces w Linuksie nie sprawdza w pętli, czy ktoś chce, żeby się zakończył. Zamiast tego jądro może go **przerwać** w dowolnym momencie i doręczyć mu jednobajtowy komunikat: „stało się coś, na co masz zareagować". To jest sygnał — nie niesie żadnych danych poza swoim numerem, jest tylko informacją „zdarzenie numer N".

Mechanizm powstał, bo potrzebne było uniwersalne, asynchroniczne API do trzech rzeczy naraz: powiadamiania procesu o zdarzeniach systemowych (dziecko się zakończyło, terminal się rozłączył), przerywania go z zewnątrz (Ctrl+C, `kill`) i wymuszania jego zakończenia, kiedy grzeczne metody nie zadziałają. Stąd w tabeli sygnałów obok siebie leżą rzeczy tak różne jak „proces potomny zmienił stan" (`SIGCHLD`) i „zginiesz teraz, bez wyjątków" (`SIGKILL`).

Nazwa komendy `kill` jest myląca nieprzypadkowo — to skrót myślowy z czasów, gdy jedynym praktycznym zastosowaniem wysyłania sygnału z terminala było zabijanie zawieszonych procesów. Dziś `kill` wysyła dowolny sygnał z tabeli, a „zabij" jest tylko jednym z kilkudziesięciu możliwych znaczeń.

Analogia

Pomyśl o **systemie alarmowym w biurowcu**.

Kiedy trzeba opróżnić budynek w sposób kontrolowany, ochrona włącza komunikat: „Prosimy o opuszczenie budynku najbliższym wyjściem". Ludzie mają chwilę, żeby zamknąć dokumenty w sejfie, zapisać plik, nad którym pracowali, zgasić kuchenkę, wziąć kurtkę. Ewakuacja trwa kilkadziesiąt sekund, ale kończy się porządkiem — nic nie zostaje otwarte na oścież, nic się nie pali. To jest `SIGTERM`: komunikat, na który proces może zareagować po swojemu, zanim faktycznie zniknie.

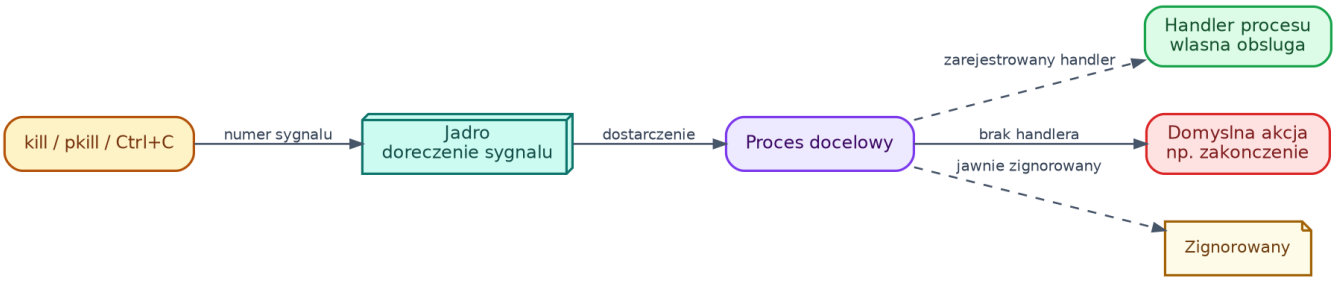
Odcięcie prądu w całym budynku to `SIGKILL`. Działa natychmiast, bez wyjątków, i nie obchodzi go, że ktoś akurat trzymał otwarte drzwi sejfu. Winda zatrzymuje się między piętrami z ludźmi w środku. Serwerownia traci zasilanie w trakcie zapisu na dysk. Jest to metoda pewna i szybka — i właśnie dlatego zarezerwowana na sytuacje, w których komunikat głosowy już nie pomoże, bo np. ktoś celowo go ignoruje.

Analogia niesie ze sobą dokładnie tę pułapkę, która dotyczy `kill -9`: odcięcie prądu nie da się cofnąć ani przechwycić przez żaden system budynku, bo cały system budynku — łącznie z głośnikami, które nadawałyby komunikat — też traci zasilanie. `SIGKILL` działa tak samo: dociera bezpośrednio do jądra, które usuwa proces ze swojej tabeli, z pominięciem jakiegokolwiek kodu samego procesu. Nie ma miejsca, w którym proces mógłby „usłyszeć" ten sygnał i coś z nim zrobić.

Jak to działa

□□ Rozwiń: jak to działa pod spodem

> **Pojęcie: obsługa sygnału.** Dla każdego sygnału jądro trzyma w tabeli procesu zapis tego, co ma się z nim stać: wywołać własny kod obsługi zarejestrowany przez program (handler), zignorować sygnał, albo zastosować domyślną akcję systemową — najczęściej zakończenie procesu. Dwa sygnały, `SIGKILL` i `SIGSTOP`, są **nieprzechwytywalne**: nigdy nie trafiają do tej tabeli, nie da się dla nich zarejestrować handlera ani ich zignorować, bo mają gwarantować, że da się zatrzymać dowolny proces — nawet taki, który celowo albo przez błąd broni się przed zamknięciem.



Jądro doręcza sygnał procesowi, a ten wykonuje własny handler, gdy go zarejestrował, albo domyślną akcję systemową, albo po prostu go ignoruje (kliknięcie otwiera pełny rozmiar)

Dostarczenie sygnału. Kiedy przychodzi sygnał, jądro sprawdza tabelę procesu i albo przerywa jego normalny bieg, żeby wywołać zarejestrowany handler, albo od razu stosuje domyślną akcję — zakończenie procesu, zakończenie ze zrzutem pamięci (`core dump`), zatrzymanie albo wznowienie, zależnie od konkretnego sygnału.

Dlaczego SIGKILL i SIGSTOP są wyjątkiem. To jest powód, dla którego `kill -9` powinien być ostatecznością. Proces zabity `SIGTERM`, jeśli ma do tego kod, może w swoim handlerze dopisać niezapisane dane na dysk, zamknąć transakcję w bazie, usunąć plik `.lock` i plik tymczasowy, zamknąć gniazda sieciowe bez zrywania klientowi połączenia w pół odpowiedzi. Proces zabity `SIGKILL` nie robi nic z tego — jądro usuwa go ze swojej tabeli dokładnie w tym miejscu, w którym się znajdował, z otwartym plikiem tymczasowym i niedopisaną transakcją. Stąd reguła, którą stosuje też `systemd` przy zatrzymywaniu usług: najpierw `SIGTERM`, odczekanie na rozsądny czas, a dopiero potem `SIGKILL`. `timeout -k` w Fiszce robi to w jednej komendzie.

SIGHUP i jego historia. Nazwa rozwija się jako „hang up" — rozłączenie linii telefonicznej, dosłownie odłożenie słuchawki. W czasach terminali podłączonych fizycznym łączem szeregowym, kiedy to łącze się zrywało, jądro wysyłało `SIGHUP` do wszystkich procesów tej sesji terminala, żeby posprzątały i się zakończyły — bo proces bez terminala i tak nie miał już z kim rozmawiać. Ten mechanizm opisuje szerzej strona *Zadania powłoki: jobs, bg, fg, nohup, disown*.

Z czasem `SIGHUP` zyskał drugie, zupełnie inne, umowne znaczenie u demonów: „przeładuj plik konfiguracyjny, ale nie kończ procesu". Nie ma w tym żadnej magii jądra — to konwencja, którą programiści demonów sami sobie wymyślili, bo `SIGHUP` i tak był sygnałem, którego proces bez terminala normalnie nigdy nie dostawał w innym kontekście, więc nadawał się na wolny numer do zagospodarowania. `nginx`, `sshd`, `rsyslog` i większość demonów systemowych obsługują `SIGHUP` w ten drugi sposób. Trzeba sprawdzić dokumentację konkretnej usługi — nie ma gwarancji, że akurat ta aplikacja to zaimplementowała.

Krok po kroku

1. Sprawdź, jaki numer ma dany sygnał.

```
kill -l
```

Numery 1–19 (poza drobnymi odchyleniami dla `SIGSTKFLT`) są ustabilizowane na Linuksie od dekad i takie samo `kill -9` zadziała na RHEL-u, Debianie i SLES-ie. Nie polegaj jednak na numerach w skryptach przenośnych między architekturami — na SPARC czy Alpha kolejność bywa inna. Pisz `kill -TERM`, nie `kill -15`, jeśli skrypt ma kiedykolwiek trafić gdzie indziej niż x86_64/ARM.

2. Wyślij SIGTERM, poczekaj, dobij SIGKILL — ręcznie i jedną komendą.

```
kill -TERM 4821
sleep 10
if kill -0 4821 2>/dev/null; then
    echo "proces 4821 nadal żyje po 10s, dobijam"
    kill -KILL 4821
fi
```

`kill -0` nie wysyła żadnego realnego sygnału — jądro tylko sprawdza uprawnienia i istnienie PID-u. To samo jedną komendą:

```
timeout -k 5 10 dlugi_import.sh
```

Po 10 sekundach `timeout` wysyła `SIGTERM`; jeśli proces nie skończy się w kolejne 5 sekund, dobija go `SIGKILL`. Kod wyjścia `124` oznacza przekroczenie limitu — przydatne w skryptach, które mają to rozróżnić od normalnego zakończenia.

3. Przeładuj konfigurację demona bez przerywania jego pracy.

```
sudo kill -HUP $(cat /run/nginx.pid)
```

albo, na systemie z `systemd`, bezpieczniej:

```
sudo systemctl kill -s HUP nginx
sudo systemctl reload nginx        # dla usług, które to wspierają — woła to samo pod spodem
```

`systemctl reload` jest bezpieczniejsze niż ręczne `kill -HUP`, bo `systemd` wie dokładnie, którym PID-em jest dana usługa (bez ryzyka trafienia sygnałem w niewłaściwy proces) i sprawdzi, czy jednostka w ogóle deklaruje `ExecReload`.

4. Znajdź proces po nazwie i zabij go bezpiecznie.

```
pgrep -af 'worker.py'
```

```
14821 python3 /opt/app/worker.py --queue=default
14822 python3 /opt/app/worker.py --queue=priority
```

Zobacz najpierw, co dokładnie złapiesz — dopiero potem zawężaj i zabijaj:

```
pkill -TERM -f 'worker.py --queue=default'
```

Wysyła `SIGTERM` tylko do procesu z `--queue=default`, zostawiając `--queue=priority` przy życiu.

5. Zabij po dokładnej nazwie polecenia zamiast wzorca.

```
killall -TERM nginx
```

`killall` dopasowuje do **nazwy polecenia** (`comm`, do 15 znaków w jądrze), nie do dowolnego fragmentu linii poleceń jak `pkill -f`. Jest bardziej precyzyjne w tym sensie, ale ma swoją własną pułapkę opisaną niżej.

6. Posprzątaj plik tymczasowy niezależnie od tego, jak skrypt się skończy.

```
#!/bin/bash
TMPFILE=$(mktemp /tmp/import.XXXXXX)
trap 'rm -f "$TMPFILE"' EXIT

echo "dane robocze" > "$TMPFILE"
# ... długie przetwarzanie ...
```

`trap` na `EXIT` uruchamia się zawsze — przy normalnym zakończeniu skryptu, przy `exit` wywołanym w środku, przy `Ctrl+C` (który wysyła `SIGINT`), a nawet przy błędzie, jeśli skrypt ma ustawione `set -e`. Plik tymczasowy zniknie, nawet jeśli ktoś przerwie skrypt w połowie.

7. Zareaguj na przerwanie z klawiatury inaczej niż na zwykłe zakończenie.

```
#!/bin/bash
trap 'echo "przerwano, kończę"; exit 1' INT TERM

echo "pracuję, przerwij mnie przez Ctrl+C"
sleep 60
```

Uruchomienie i przerwanie w trakcie działania:

```
pracuję, przerwij mnie przez Ctrl+C
^C
przerwano, kończę
```

Bez tego `trap` u `Ctrl+C` po prostu zakończyłby skrypt domyślną akcją `SIGINT` — bez żadnego komunikatu i bez szansy posprzątać.

Częste błędy

`kill -9` jako pierwszy odruch, zamiast ostateczności.

```
$ kill -9 4821
```

Proces znika natychmiast, więc problem wygląda na rozwiązany. Ale jeśli był to serwer bazy danych w trakcie zapisu transakcji albo skrypt trzymający uchwyt do pliku tymczasowego — nic nie zostało posprzątane, bo `SIGKILL` nie daje procesowi ani jednej instrukcji procesora na reakcję. Efekt widzisz dopiero przy kolejnym starcie: plik `.lock` blokujący ponowne uruchomienie, plik tymczasowy w `/tmp`, baza uruchamiająca odzyskiwanie po awarii zamiast normalnego startu. Zasada: najpierw `kill -TERM` (albo bez flagi, bo `TERM` jest domyślne), odczekaj kilka–kilkanaście sekund, dopiero potem `kill -9`.

`pkill -f wzorzec` zabija więcej, niż zamierzałeś.

```
$ pkill -f python
```

Wzorzec `python` w trybie `-f` dopasowuje się do **całej linii poleceń** każdego procesu w systemie. Jeśli na maszynie działają jednocześnie `python3 /opt/app/worker.py` i `python3 /usr/bin/dnf` (na RHEL-u `dnf` samo jest napisane w Pythonie) — obydwa padają naraz, bo każdy ma `python` gdzieś w swojej linii poleceń. Zawsze najpierw:

```
pgrep -af python
```

Zobacz pełną listę dopasowań, zawęź wzorzec (`--queue=default` zamiast `python`), i dopiero wtedy wywołaj `pkill` z tym samym, sprawdzonym wzorcem.

Proces w tle bez przekierowania wyjścia — i sygnał `SIGTTIN`.

```
$ dlugie_zadanie.sh &
[1] 14930
```

Skrypt wypisuje coś na ekran w środku Twojej pracy w terminalu, a jeśli w pewnym momencie spróbuje coś przeczytać ze standardowego wejścia (np. pyta o potwierdzenie), dostanie od powłoki sygnał `SIGTTIN` i zawiśnie w stanie `T` (zatrzymany) bez żadnego komunikatu o błędzie — po prostu przestaje działać, a `jobs` pokaże go jako `Stopped`. Pełne omówienie zadań powłoki i przekierowań w tle jest na stronie *Zadania powłoki: jobs, bg, fg, nohup, disown*; sama mechanika przekierowań na stronie *Przekierowania: >, >>, 2>, &>, <, tee*.

Zagnieżdżony `pkill`/`killall` bez `-u`, trafiający w cudze procesy na współdzielonym serwerze.

```
$ killall -9 java
```

Na serwerze, na którym kilku administratorów uruchamia własne procesy Javy (jedna instancja Tomcata, jeden proces Elasticsearcha, czyjś prywatny skrypt testowy), to polecenie ubija je wszystkie naraz — `killall` bez `-u` nie rozróżnia właściciela procesu, tylko samą nazwę binarki. Zawężaj zawsze użytkownikiem, jeśli działasz na maszynie współdzielonej:

```
killall -u tomcat -TERM java
```

Oczekiwanie, że sygnał obudzi proces w stanie D.

```
$ kill -9 22104
$ ps -p 22104 -o stat=
D
```

Proces w nieprzerywalnym oczekiwaniu na I/O jądra (stan `D`) ignoruje nawet `SIGKILL` — sygnał zostaje zapamiętany, ale obsłużony dopiero, gdy proces sam wyjdzie z operacji jądra, a jeśli ta się zawiesiła (martwy NFS, padający dysk), to może nie nastąpić nigdy z poziomu userspace. Pełne wyjaśnienie mechanizmu i co wtedy robić: *Stany procesów: zombie, sieroty, D-state*.

□□ Ćwiczenia

>

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

Ćwiczenie 1 — trap i sprzątanie po sygnale

```
cd /tmp

# 1. Napisz skrypt, który przechwytuje sygnały i sprząta po sobie
cat > demo_trap.sh <<'EOF'
#!/bin/bash
TMPFILE=$(mktemp /tmp/demo.XXXXXX)
echo "utworzyłem plik tymczasowy: $TMPFILE"

sprzataj() {
    echo "dostałem sygnał, usuwam $TMPFILE"
    rm -f "$TMPFILE"
    exit 0
}
trap sprzataj INT TERM EXIT

echo "śpię 60 sekund, przerwij mnie przez Ctrl+C albo kill z innej sesji (PID: $$)"
sleep 60
EOF
chmod +x demo_trap.sh
./demo_trap.sh
```

W drugiej sesji terminala sprawdź PID i wyślij sygnał:

```
pgrep -af demo_trap
kill -TERM $(pgrep -f demo_trap.sh)
```

Wróć do pierwszej sesji — skrypt powinien wypisać komunikat sprzątający i zakończyć się sam, a plik tymczasowy zniknąć. Sprawdź:

```
ls -la /tmp/demo.*
```

Czego się spodziewać

Po `kill -TERM` skrypt wypisze `dostałem sygnał, usuwam ...` i zakończy działanie. Plik `demo.XXXXXX` nie będzie już istniał — `ls` zwróci `No such file or directory`. Powtórz eksperyment z `Ctrl+C` zamiast `kill` z drugiej sesji — efekt będzie identyczny, bo `Ctrl+C` też wysyła `SIGINT`, który skrypt przechwytuje tą samą funkcją `sprzataj`.

Jeśli zamiast `kill -TERM` użyjesz `kill -9`, `trap` się NIE wykona — bo `SIGKILL` nie da się przechwycić. Plik tymczasowy zostanie na dysku. Warto to zobaczyć na własne oczy, żeby zapamiętać różnicę.

Pytania kontrolne

1. Dlaczego `kill -9` zawsze powinien być ostatnią próbą, a nie pierwszą?

Odpowiedź

Bo `SIGKILL` nie daje procesowi żadnej szansy na reakcję — jądro usuwa go natychmiast, z pominięciem jakiegokolwiek kodu aplikacji. Proces nie zdąży zamknąć pliku, dopisać transakcji do bazy, zwolnić blokady ani usunąć pliku tymczasowego. `SIGTERM` daje mu szansę zrobić to wszystko we własnym handlerze, zanim faktycznie się zakończy. Dopiero jeśli proces nie zareaguje na `SIGTERM` w rozsądnym czasie, sięga się po `SIGKILL`.

2. Wpisujesz `pkill -f worker`, a razem z procesem, którego szukałeś, pada też Twoja sesja SSH i skrypt monitorujący. Co poszło nie tak?

Odpowiedź

`-f` dopasowuje wzorzec do całej linii poleceń, nie do samej nazwy programu. Jeśli linia poleceń sesji SSH albo skryptu monitorującego zawiera gdziekolwiek napis „worker" (np. w argumencie, ścieżce albo nazwie hosta), `pkill -f` też je złapie. Trzeba było najpierw uruchomić `pgrep -af worker`, zobaczyć pełną listę dopasowań i dopiero potem zawęzić wzorzec tak, żeby pasował wyłącznie do zamierzonego procesu.

3. Czym różni się `SIGSTOP` od `SIGTERM` w kontekście tego, czy proces może to zignorować?

Odpowiedź

`SIGTERM` można przechwycić własnym handlerem, zignorować, albo pozwolić na domyślne zakończenie procesu — decyduje o tym kod aplikacji. `SIGSTOP` (podobnie jak `SIGKILL`) nie da się ani przechwycić, ani zignorować — jądro zawsze zatrzyma proces bezwarunkowo. Różnica względem `SIGKILL` jest taka, że `SIGSTOP` da się cofnąć sygnałem `SIGCONT`, a `SIGKILL` jest nieodwracalny, bo proces po nim przestaje istnieć.

4. Proces kończy się kodem wyjścia `137`. Co to oznacza i jak to policzyłeś?

Odpowiedź

`137 = 128 + 9`, a `9` to numer `SIGKILL` — proces został zabity tym sygnałem, nie zakończył się sam przez `exit()`. Konwencja `128 + numer sygnału` obowiązuje dla wszystkich procesów zabitych sygnałem; pełne omówienie kodów wyjścia jest na stronie *Kod wyjścia, $?, &&, ||, średnik*.

Powiązane: ps: zdjęcie stanu procesów · Stany procesów: zombie, sieroty, D-state · Zadania powłoki: jobs, bg, fg, nohup, disown · Kod wyjścia, $?, &&, ||, średnik · Przekierowania: >, >>, 2>, &>, <, tee

Źródła: `man 1 kill` · `man 1 pkill` · `man 1 pgrep` · `man 1 killall` · `man 1 timeout` · `man 7 signal` · `man 1 trap` (bash builtins)

Revision #18
Created 2026-09-17 15:07:16 UTC by Tomasz Konieczny
Updated 2026-10-03 19:08:15 UTC by Tomasz Konieczny