# Obciążenie systemu: load average, uptime, dmesg

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

*Trzy liczby z `uptime` to najczęściej cytowany i najczęściej źle rozumiany wskaźnik w Linuksie. Bez odniesienia do liczby rdzeni nic nie znaczą — a jedno z założeń liczenia obciążenia w Linuksie różni się od reszty rodziny Uniksów, o czym większość administratorów nie wie.*

### Skróty użyte na tej stronie (6)

| Skrót | Co znaczy |
|---|---|
| **BSD** | Berkeley Software Distribution: rodzina Uniksów z uczelni Berkeley, inna gałąź niż Linux, stąd czasem inna składnia tych samych poleceń |
| **CPU** | Central Processing Unit: procesor |
| **OOM** | Out Of Memory: brak pamięci; OOM killer to mechanizm jądra zabijający proces, gdy jej zabraknie |
| **PID** | Process ID: numer procesu w systemie |
| **POSIX** | Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach |
| **RAM** | Random Access Memory: pamięć operacyjna |

# Fiszka

**TYLKO ODCZYT**

## Sprawdzenie czasu działania i trzech liczb obciążenia: uptime

```
uptime
```

Pokazuje aktualny czas, jak długo system działa bez restartu, liczbę zalogowanych sesji i średnie obciążenie za 1, 5 i 15 minut.

Przykład:

```
 14:32:07 up 23 days,  4:12,  3 users,  load average: 2.15, 1.98, 1.76
```

**TYLKO ODCZYT**

## Wypisanie surowych danych, z których liczy uptime: cat /proc/loadavg

```
cat /proc/loadavg
```

Piąte pole to licznik ostatniego uruchomionego procesu (PID) — poza tym to dokładnie te same trzy średnie plus liczba aktualnie uruchomionych/wszystkich procesów.

Przykład:

```
2.15 1.98 1.76 3/512 48210
```

`3/512` znaczy: 3 procesy aktualnie w stanie uruchomionym lub gotowym (`runnable`), spośród 512 wszystkich procesów i wątków w systemie.

**PID** = Process ID: numer procesu w systemie · **cat** = concatenate: sklej i wypisz pliki

**TYLKO ODCZYT**

## Sprawdzenie, kto jest zalogowany, i obciążenia systemu: w

```
w
```

Łączy funkcję `uptime` (nagłówek z load average) z funkcją `who` (lista sesji) — jedno polecenie zamiast dwóch, gdy potrzebujemy obu informacji naraz.

Przykład:

```
 14:32:07 up 23 days,  4:12,  3 users,  load average: 2.15, 1.98, 1.76
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
adm_user pts/0    10.20.20.15      09:12    0.00s  0.08s  0.02s w
root     pts/1    10.20.20.22      08:40    3:22m  0.01s  0.01s -bash
```

**TYLKO ODCZYT**

## Sprawdzenie liczby dostępnych rdzeni z perspektywy powłoki: nproc

```
nproc
```

Zwraca liczbę procesorów logicznych dostępnych *dla bieżącego procesu* — jeśli działamy w kontenerze z ograniczonym cgroupem CPU, ta liczba może być mniejsza niż fizyczna liczba rdzeni hosta.

Przykład:

```
8
```

**CPU** = Central Processing Unit: procesor

**TYLKO ODCZYT**

## Sprawdzenie pełnego opisu procesora: lscpu

```
lscpu
```

Architektura, liczba rdzeni fizycznych i logicznych (z uwzględnieniem hyperthreadingu), liczba gniazd, częstotliwości — cały kontekst potrzebny, żeby liczba `nproc` w ogóle miała sens.

Przykład:

```
Architecture:            x86_64
CPU(s):                  8
On-line CPU(s) list:     0-7
Thread(s) per core:      2
Core(s) per socket:      4
Socket(s):               1
Model name:              Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz
```

**lscpu** = list CPU: wypisuje budowę procesora: gniazda, rdzenie, wątki i węzły NUMA

**TYLKO ODCZYT**

## Sprawdzenie liczby rdzeni zgodnie z POSIX: getconf _NPROCESSORS_ONLN

```
getconf _NPROCESSORS_ONLN
```

Standard POSIX, obecny na każdym systemie uniksowym, nawet tam, gdzie `nproc` (z pakietu GNU coreutils) mógłby nie być zainstalowany — przydatne w przenośnych skryptach.

Przykład:

```
8
```

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

**TYLKO ODCZYT**

## Wypisanie bufora komunikatów jądra z czytelnym czasem: dmesg -T

```
dmesg -T
```

`-T` tłumaczy znaczniki czasu jądra (sekundy od startu) na pełne daty. Pierwsze miejsce, gdzie szukać przyczyny nagłego skoku obciążenia — np. błędu sprzętowego albo problemu z dyskiem.

Przykład:

```
[Wed Sep 17 03:12:41 2026] EXT4-fs (dm-0): error count since last fsck: 3
[Wed Sep 17 03:15:02 2026] ata1.00: exception Emask 0x0 SAct 0x0 SErr 0x0 action 0x0
```

**dmesg** = pokaż komunikaty jądra, przydatne przy diagnozowaniu sprzętu i dysków

**TYLKO ODCZYT**

## Nasłuchiwanie nowych komunikatów jądra na żywo: dmesg -w

```
dmesg -w
```

Zamiast jednorazowego zrzutu, `-w` (watch/follow) zostawia polecenie działające i wypisuje każdy nowy komunikat jądra w momencie jego pojawienia się — odpowiednik `tail -f` dla logu jądra.

**dmesg** = pokaż komunikaty jądra, przydatne przy diagnozowaniu sprzętu i dysków

**TYLKO ODCZYT**

## Wypisanie tylko poważniejszych komunikatów: dmesg -l err,warn

```
dmesg -l err,warn
```

`-l` filtruje po poziomie ważności (priority) — przy dużym, zaśmieconym buforze pozwala pominąć informacyjny szum i zobaczyć wyłącznie błędy i ostrzeżenia.

Przykład:

```
[Wed Sep 17 03:12:41 2026] EXT4-fs (dm-0): error count since last fsck: 3
```

**dmesg** = pokaż komunikaty jądra, przydatne przy diagnozowaniu sprzętu i dysków

**TYLKO ODCZYT**

## Sprawdzenie komunikatów jądra z dziennika systemd: journalctl -k -p err

```
journalctl -k -p err
```

`-k` ogranicza wynik do komunikatów jądra (kernel), `-p err` do poziomu błąd i poważniejszych. Przetrwa restart, jeśli journal jest skonfigurowany trwale — w przeciwieństwie do bufora `dmesg`.

**journalctl** = journal control: czytanie dziennika systemd

**TYLKO ODCZYT**

## Wypisanie komunikatów jądra z poprzedniego rozruchu: journalctl -k -b -1

```
journalctl -k -b -1
```

`-b -1` wybiera poprzedni rozruch systemu (`-b -2` przedostatni, `-b` bez liczby bieżący). Jedyny sposób zajrzenia do logu jądra sprzed ostatniego restartu, jeśli `dmesg` (którego bufor czyści się przy każdym starcie) już dawno nie ma tych danych.

□□ **journalctl** = journal control: czytanie dziennika systemd

□□ TYLKO ODCZYT

## Sprawdzenie presji CPU (PSI): cat /proc/pressure/cpu

```
cat /proc/pressure/cpu
```

Nowocześniejsza miara niż load average — mówi wprost, jaki procent czasu przynajmniej jeden proces (`some`) albo wszystkie procesy naraz (`full`) czekały na dostęp do CPU, licząc osobno okna 10 sekund, 1 minuty i 5 minut.

Przykład:

```
some avg10=2.15 avg60=1.80 avg300=1.20 total=184213045
```

□□ **CPU** = Central Processing Unit: procesor · **cat** = concatenate: sklej i wypisz pliki

□□ TYLKO ODCZYT

## Sprawdzenie presji wejścia/wyjścia (PSI): cat /proc/pressure/io

```
cat /proc/pressure/io
```

To samo podejście zastosowane do I/O — procent czasu, w którym procesy czekały na operacje dyskowe zamiast mogąc kontynuować pracę.

Przykład:

```
some avg10=0.40 avg60=0.55 avg300=0.30 total=52014402
full avg10=0.10 avg60=0.20 avg300=0.15 total=18022310
```

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

□□ TYLKO ODCZYT

## Sprawdzenie presji pamięci (PSI): cat /proc/pressure/memory

```
cat /proc/pressure/memory
```

Analogicznie dla pamięci — wysoki `full` tutaj to znacznie bardziej wiarygodny sygnał zbliżającego się OOM killera niż samo `available` z `free`.

Przykład:

```
some avg10=0.00 avg60=0.00 avg300=0.00 total=0
full avg10=0.00 avg60=0.00 avg300=0.00 total=0
```

**OOM** = Out Of Memory: brak pamięci; OOM killer to mechanizm jądra zabijający proces, gdy jej zabraknie · **cat** = concatenate: sklej i wypisz pliki

**TYLKO ODCZYT**

## Sprawdzenie daty i godziny ostatniego rozruchu: uptime -s

```
uptime -s
```

Sam moment startu systemu, w formacie gotowym do dalszego przetwarzania w skrypcie (`date -d`, obliczenia różnicy czasu).

Przykład:

```
2026-08-25 10:20:12
```

**TYLKO ODCZYT**

## Sprawdzenie ostatniego rozruchu innym narzędziem: who -b

```
who -b
```

Alternatywne źródło tej samej informacji — przydatne, gdy skrypt już i tak używa `who` do innych celów i nie chce dokładać zależności od `uptime`.

Przykład:

```
         system boot  2026-08-25 10:20
```

**TYLKO ODCZYT**

### Wszystkie flagi

| Polecenie | Co pokazuje |
|---|---|
| `uptime` / `cat /proc/loadavg` | trzy średnie obciążenia (1/5/15 min) |
| `nproc` / `lscpu` / `getconf _NPROCESSORS_ONLN` | liczba rdzeni, kontekst do interpretacji load average |
| `dmesg -T` / `dmesg -w` / `dmesg -l err,warn` | bufor komunikatów jądra |
| `journalctl -k -p err` / `-b -1` | to samo z trwałego dziennika systemd |
| `/proc/pressure/{cpu,io,memory}` | PSI — nowocześniejsza miara zamiast samego load average |

> **Uważaj**
>
> - Load average bez odniesienia do `nproc` nic nie znaczy — 8 na maszynie 16-rdzeniowej to spokój, 8 na maszynie 2-rdzeniowej to alarm.
> - Wysoki load przy niskim zużyciu CPU (widocznym np. w `vmstat` / `top`) to zwykle nie CPU, tylko czekanie na I/O.
> - `dmesg` bez `-T` pokazuje sekundy od startu, nie daty — łatwo pomylić czas zdarzenia przy korelacji z innymi logami.

**CPU** = Central Processing Unit: procesor

# Z życia wzięte

**Z ŻYCIA WZIĘTE**

Poniedziałek rano. Serwer k3s01 pokazuje load average dwadzieścia przy zaledwie czterech rdzeniach, ale procesor w htopie stoi bezczynnie. Tomek tłumaczy Oli, że większość procesów czeka w stanie D na odpowiedź wolnego dysku sieciowego, nie na czas procesora, więc wysoki load average w tym wypadku nie znaczy przeciążonego CPU.

# Po co to istnieje

> **Pojęcie: średnie obciążenie (load average).** Trzy liczby opisujące, ile procesów średnio czekało na dostęp do procesora (albo znajdowało się w stanie nieprzerwalnego oczekiwania — patrz niżej) w oknach czasowych 1, 5 i 15 minut. Nie jest to procent wykorzystania CPU —

to jest liczba *żądań* na procesor uśredniona w czasie, więc jej sensowna interpretacja wymaga wiedzy, ile ten system w ogóle ma procesorów do zaoferowania.

**Pojęcie: kolejka gotowych do uruchomienia.** Gdy procesów chcących w danej chwili wykonywać obliczenia jest więcej niż dostępnych rdzeni CPU, nadmiarowe procesy czekają w kolejce, gotowe do uruchomienia w każdej chwili, gdy tylko któryś rdzeń się zwolni. To jest kolumna `r` w `vmstat` i główny składnik load average — im dłuższa ta kolejka średnio w czasie, tym wyższa liczba.

Administrator zerka na load average z jednego prostego powodu: to jest najszybszy, jednowierszowy sposób oceny, czy system jest obciążony, bez uruchamiania niczego cięższego niż `uptime`. Problem w tym, że sama liczba, bez kontekstu liczby rdzeni, jest praktycznie bezużyteczna — i to jest treść tej strony.

# Analogia

Wyobraź sobie **kasy w supermarkecie i kolejkę klientów**.

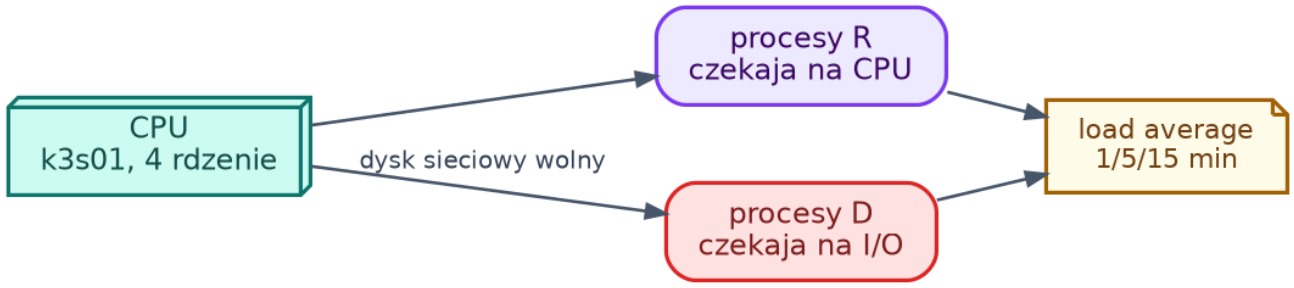
Load average to średnia liczba klientów czekających w kolejkach do wszystkich kas razem wziętych, uśredniona w czasie. Ale ta liczba nic nie mówi, dopóki nie wiesz, ile kas jest w ogóle otwartych. Dziesięciu czekających klientów przy dwudziestu czynnych kasach to sklep pusty — każda kasa ma zapas mocy. Ci sami dziesięciu klientów przy dwóch czynnych kasach to korek sięgający do wejścia.

Load average = 8 to dokładnie ta sama liczba w obu scenariuszach — dopiero podzielenie jej przez liczbę „kas" (rdzeni CPU, `nproc`) mówi, czy to spokojny wtorek, czy przedświąteczny szczyt. I tak jak w supermarkecie kolejka bywa długa nie dlatego, że kasjerki wolno skanują towar, tylko dlatego że jedna z kas utknęła na sprawdzaniu ceny (odpowiednik czekania na wolny dysk) — wysoki load przy leniwym procesorze to zwykle sygnał, że wąskim gardłem jest coś innego niż sama moc obliczeniowa.

# Jak to działa

**Rozwiń: jak to działa pod spodem**

**Pojęcie: nieprzerwalny sen wliczany do obciążenia.** Tu Linux różni się od reszty rodziny systemów uniksowych w sposób, który zaskakuje nawet doświadczonych administratorów: w klasycznym BSD i innych Uniksach load average liczy wyłącznie procesy czekające na CPU (stan `R` — running/runnable). Linux dolicza do tego również procesy w stanie **D** — nieprzerwalnym śnie (`uninterruptible sleep`), czyli czekające na zakończenie operacji I/O na poziomie jądra, których nie da się przerwać nawet sygnałem `SIGKILL`. Konsekwencja: na Linuksie load average może być wysokie *bez żadnego obciążenia procesora* — wystarczy, że kilka procesów utknęło na wolnym dysku sieciowym albo przeciążonym storage'u. Administrator patrzący wyłącznie na `top` / `vmstat` i widzący niski `%us` / `%sy` przy wysokim load bywa zdezorientowany, dopóki nie sprawdzi kolumny `wa` (I/O wait) i procesów w stanie `D` (`ps aux | awk '$8 ~ /D/'`) — pełny opis stanu `D` jest na stronie o stanach procesów w rozdziale o procesach.




*Load average liczy razem procesy czekające na procesor i procesy czekające na wolny dysk. (kliknięcie otwiera pełny rozmiar)*

Same trzy liczby (1/5/15 minut) to wykładniczo ważone średnie ruchome (exponentially weighted moving average) liczone przez jądro co 5 sekund na podstawie liczby procesów w stanach `R` i `D` w danym momencie — nie jest to prosta średnia arytmetyczna z ostatnich N minut, tylko formuła nadająca większą wagę świeższym próbkom. Dlatego load average reaguje na nagły skok obciążenia szybciej, niż mogłoby się wydawać, patrząc tylko na etykietę „15 minut".

**Pojęcie: bufor komunikatów jądra.** Jądro Linuksa pisze własne komunikaty diagnostyczne (błędy sprzętowe, ostrzeżenia sterowników, zdarzenia systemu plików) do cyklicznego bufora w pamięci, dostępnego przez `dmesg` albo `/dev/kmsg`. Bufor ma ograniczony rozmiar i czyści się przy każdym restarcie — jeśli komunikat jest istotny na dłużej niż do najbliższego rebootu, `journalctl -k` (o ile journal jest skonfigurowany trwale) to jedyne miejsce, gdzie na pewno przetrwa.

> **Pojęcie: PSI (Pressure Stall Information).** Mechanizm dodany w jądrze 4.20 (2018), dostępny przez `/proc/pressure/{cpu,io,memory}`, uznawany za uczciwszą i bardziej precyzyjną miarę niż klasyczne load average. Zamiast jednej zlepionej liczby (CPU plus I/O plus, pośrednio, pamięć — bo procesy czekające na swap też trafiają do stanu `D`), PSI rozdziela presję na trzy niezależne zasoby i dla każdego liczy dwa warianty: `some` (przynajmniej jeden proces czekał) i `full` (wszystkie uruchamialne procesy czekały naraz, czyli CPU/dysk/pamięć były realnie zablokowane dla całego systemu, nie tylko dla jednego procesu). `full` bliskie zeru przy wysokim `some` oznacza, że problem dotyka pojedynczych procesów, a reszta systemu radzi sobie normalnie — rozróżnienie, którego load average w ogóle nie oferuje.

# Krok po kroku

**1. Odczytaj load average razem z liczbą rdzeni — nigdy osobno.**

```
uptime
nproc
```

```
 14:32:07 up 23 days,  4:12,  3 users,  load average: 6.80, 5.20, 4.10
8
```

Load 6.8 na 8 rdzeniach to system pracujący blisko granicy, ale wciąż w zasięgu — mniej niż 1.0 na rdzeń. Ta sama liczba 6.8 na maszynie z `nproc` równym 2 oznaczałaby kolejkę ponad trzykrotnie dłuższą niż liczba dostępnych procesorów.

**2. Sprawdź, czy rosnący trend (1 min > 5 min > 15 min) to CPU, czy I/O.**

```
vmstat 2 5
```

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  5  524288 2100000 184000 5100000    0    0  8200  1200  180  140 3  2 15 80  0
```

Kolumna `b` (procesy w nieprzerwalnym śnie) równa 5, a `wa` (czekanie na I/O) 80% przy `us`+`sy` zaledwie 5% — obciążenie widoczne w `uptime` pochodzi niemal wyłącznie z czekania na dysk, nie z realnej pracy procesora.

**3. Dla dokładniejszego obrazu sięgnij po PSI zamiast zgadywać z load average.**

```
cat /proc/pressure/io
cat /proc/pressure/cpu
```

```
some avg10=78.40 avg60=65.20 avg300=40.10 total=982104200
full avg10=71.20 avg60=58.90 avg300=35.40 total=871029400
some avg10=4.10 avg60=3.80 avg300=3.20 total=52014402
full avg10=0.20 avg60=0.15 avg300=0.10 total=1802331
```

`io` z `full avg10` na poziomie 71% mówi wprost: przez ostatnie 10 sekund system spędził 71% czasu z *wszystkimi* uruchamialnymi procesami zablokowanymi na I/O naraz — to jednoznacznie potwierdza, że problem leży w dysku, nie w CPU (`cpu` `full` to zaledwie 0.2%).

**4. Zajrzyj do bufora jądra pod kątem błędów sprzętowych, które mogłyby to tłumaczyć.**

```
dmesg -T -l err,warn | tail -5
```

```
[Wed Sep 17 14:20:11 2026] blk_update_request: I/O error, dev sda, sector 48210200
```

Błąd I/O na konkretnym sektorze konkretnego dysku — to już wystarczający trop do dalszej diagnostyki sprzętu, nie samego systemu operacyjnego.

# Częste błędy

**Ocena load average bez sprawdzenia liczby rdzeni.**

```
$ uptime
load average: 12.40, 11.80, 10.20
```

Sama liczba 12.4 wygląda dramatycznie. Na maszynie z 32 rdzeniami (`nproc` → `32`) to wciąż spory zapas mocy. Naprawa: zawsze parą z `nproc` — reguła kciuka to porównanie load average do liczby rdzeni, nie do jakiejś stałej progowej wartości uniwersalnej dla wszystkich maszyn.

**Traktowanie wysokiego load jako dowodu przeciążenia CPU.**

Administrator widzi load 9 na maszynie 8-rdzeniowej i od razu szuka procesu żarłocznego procesorowo (`top` posortowane po `%CPU`). Jeśli prawdziwą przyczyną jest przeciążony storage (procesy w stanie `D`), taki `top` pokaże niskie zużycie CPU u wszystkich procesów i administrator traci czas, szukając czegoś, czego tam nie ma. Naprawa: sprawdź `vmstat` (kolumny `b`, `wa`) albo `/proc/pressure/io`, zanim założysz, że problem dotyczy procesora.

**Korzystanie z `dmesg` bez `-T` do korelacji czasowej z innymi logami.**

```
$ dmesg | tail -3
[1980234.552011] blk_update_request: I/O error, dev sda, sector 48210200
```

Liczba w nawiasie to sekundy od startu systemu — bez przeliczenia trudno powiązać ten wpis z konkretną minutą w logu aplikacji czy w `journalctl`. Naprawa: `dmesg -T` zawsze, gdy korelujesz czas zdarzenia z czymkolwiek innym.

**Interpretacja `/proc/loadavg` na systemie z ograniczeniami cgroup (kontenery) jako obciążenia widocznego procesowi.**

Wewnątrz kontenera `cat /proc/loadavg` zwykle pokazuje load average **całego hosta**, nie samego kontenera — jądro nie izoluje tego pliku per przestrzeń nazw (namespace) tak, jak izoluje np. widok procesów. Administrator widzi load 40 wewnątrz kontenera i panikuje, mimo że sam kontener ma limit `cpu.max` na pół rdzenia i działa spokojnie — po prostu odczytuje statystykę całej maszyny fizycznej. Naprawa: wewnątrz kontenerów licz obciążenie z `/sys/fs/cgroup/cpu.pressure` (PSI w wersji cgroup-aware), nie z `/proc/loadavg`.

**Mylenie `journalctl -k -b -1` z brakiem logu, gdy system miał tylko jeden rozruch.**

```
$ journalctl -k -b -1
No journal files were found.
```

Na świeżo zainstalowanej maszynie albo takiej, która nigdy nie była restartowana od instalacji, po prostu nie istnieje „poprzedni" rozruch do pokazania — to nie jest błąd konfiguracji journala, tylko naturalny brak historii. Naprawa: `journalctl --list-boots` pokazuje, ile rozruchów w ogóle jest zarejestrowanych, zanim spróbujesz odpytać nieistniejący.

# □□ Ćwiczenia

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

# Ćwiczenie 1 — wygenerowanie realnego obciążenia CPU i obserwacja load average

```
nproc
for i in $(seq 1 "$(nproc)"); do
  yes > /dev/null &
```

```
done
sleep 30
uptime
# sprzątanie
kill $(jobs -p)
```

Ile w przybliżeniu powinno wynosić load average (1 min) po 30 sekundach takiego obciążenia, w relacji do wyniku `nproc`?

**Odpowiedź**

Powinno zbliżyć się do liczby zwróconej przez `nproc` — uruchomiono dokładnie tyle procesów `yes` (każdy w pętli zajmujący jeden rdzeń na 100%), ile system ma rdzeni logicznych, więc kolejka gotowych do uruchomienia procesów powinna odpowiadać w przybliżeniu tej liczbie, bez znaczącego nadmiaru.

# Ćwiczenie 2 — odczyt PSI i porównanie z `uptime` podczas tego samego testu

```
for i in $(seq 1 "$(nproc)"); do
  yes > /dev/null &
done
sleep 5
cat /proc/pressure/cpu
uptime
kill $(jobs -p)
```

Dlaczego `avg10` w `/proc/pressure/cpu` reaguje na to obciążenie widocznie szybciej niż pierwsza liczba w `uptime`?

**Odpowiedź**

Load average w `uptime` to wykładniczo ważona średnia ruchoma, która mimo świeższej wagi dla nowych próbek wciąż potrzebuje kilkudziesięciu sekund, by w pełni odzwierciedlić nagłą zmianę. PSI liczy bezpośrednio procent czasu w krótkim, dziesięciosekundowym oknie, w którym procesy faktycznie czekały — reaguje więc na gwałtowny skok obciążenia zauważalnie szybciej i bardziej stromo.

# Pytania kontrolne

**1. Serwer A ma `nproc` = 16 i load average 8. Serwer B ma `nproc` = 2 i load average 1.5. Który jest bliżej granicy swoich możliwości?**

**Odpowiedź**

Serwer B. Load 1.5 na 2 rdzeniach to 75% teoretycznej pojemności kolejki, podczas gdy load 8 na 16 rdzeniach to 50%. Sama liczba load average bez podzielenia przez `nproc` daje mylący obraz — serwer A z wyższym absolutnie load jest w rzeczywistości mniej obciążony proporcjonalnie.

**2. Dlaczego na Linuksie load average może być wysokie, mimo że `top` pokazuje niemal zerowe zużycie CPU przez wszystkie procesy?**

**Odpowiedź**

Bo Linux, inaczej niż klasyczne BSD, dolicza do load average procesy w stanie nieprzerwalnego snu (`D`) — czekające na I/O jądra, nie na procesor. Duża liczba procesów utkniętych na wolnym dysku czy przeciążonym storage sieciowym podniesie load average bez jednoczesnego wzrostu zużycia CPU.

**3. Czym różni się `some` od `full` w odczycie `/proc/pressure/io`?**

**Odpowiedź**

`some` mierzy procent czasu, w którym przynajmniej jeden uruchamialny proces czekał na I/O. `full` mierzy procent czasu, w którym *wszystkie* uruchamialne procesy czekały na I/O jednocześnie — czyli cały system był w tym momencie realnie zablokowany na dysku, a nie tylko pojedynczy proces. Wysokie `full` jest poważniejszym sygnałem niż wysokie samo `some`.

**4. Dlaczego `dmesg` po restarcie systemu nie pokaże już komunikatów sprzed restartu, a `journalctl -k -b -1` może?**

**Odpowiedź**

Bufor `dmesg` żyje w pamięci RAM i jest czyszczony (nadpisywany od zera) przy każdym starcie jądra. `journalctl`, jeśli `systemd-journald` jest skonfigurowany do trwałego przechowywania logów (`Storage=persistent`), zapisuje te same komunikaty na dysk i potrafi je odtworzyć osobno dla każdego poprzedniego rozruchu przez opcję `-b`.

**Powiązane:** Pamięć naprawdę: free i /proc/meminfo · Trendy zamiast migawek: vmstat, iostat, sar, pidstat · ps — zdjęcie procesów · Stany procesów: zombie, sieroty, D-state

**Źródła:** `man 1 uptime` · `man 5 proc` (`loadavg`, `pressure`) · `man 1 dmesg` · `man 1 journalctl` · dokumentacja jądra `Documentation/accounting/psi.rst`

---

Revision #16
Created 2026-09-17 21:43:31 UTC by Tomasz Konieczny
Updated 2026-10-03 19:09:18 UTC by Tomasz Konieczny