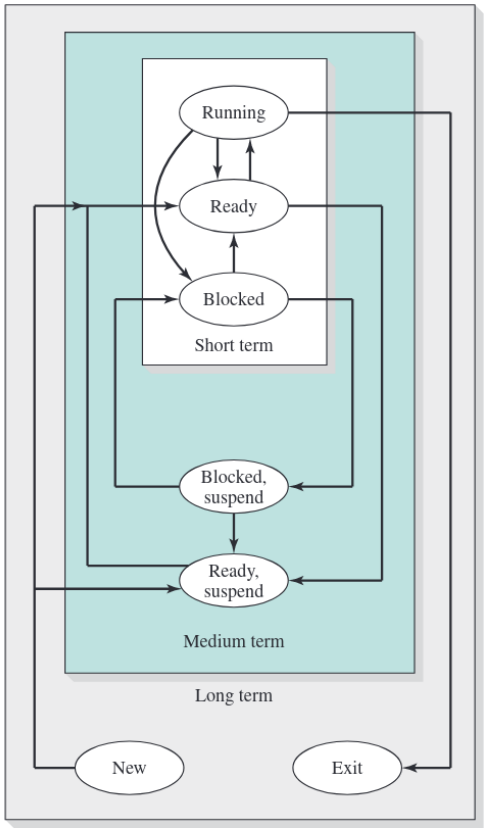
Planowanie długoterminowe, krótkoterminowe i średnioterminowe



Ostatnia aktualizacja: 14 listopada 2026

W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed., Pearson, 2018

- **planowanie długoterminowe** (*long-term scheduling*) – decyduje o tym, które nowe procesy zostaną załadowane do pamięci głównej i będą mogły konkurować o zasoby systemowe (CPU, pamięć, urządzenia wejścia-wyjścia)
- **planowanie średnioterminowe** (*medium-term scheduling*) – zarządza wymianą procesów między pamięcią główną a pamięcią wtórną (swapowaniem), co pozwala na lepsze wykorzystanie zasobów systemowych i kontrolę nad liczbą aktywnych procesów w pamięci
- **planowanie krótkoterminowe** (*short-term scheduling*) – decyduje o tym, który z gotowych do wykonania procesów otrzyma czas procesora (CPU) na wykonanie swoich instrukcji

Algorytmy przydziału procesora

Planista krótkoterminowy (ang. *scheduler*) lub **dyspozytor** – część systemu operacyjnego odpowiedzialna za przydzielanie czasu procesora w ramach przełączania zadań. Jeśli dwa lub więcej procesów znajduje się w stanie *gotowości*, o kolejności przydziału CPU decyduje planista, wykonując **algorytm szeregowania** realizujący politykę szeregowania.

Polityka szeregowania określa ogólne zasady i cele, którymi kieruje się system operacyjny przy przydzielaniu czasu procesora procesom.

Planista powinien gwarantować:

- sprawiedliwość przydziału CPU
- dobrą wydajność CPU
- mały czas odpowiedzi
- mały czas przetwarzania (*turnaround time*)
- dużą przepustowość (*throughput*)

Procesy ograniczone obliczeniowo i operacjami wejścia-wyjścia

Klasyfikacja procesów ze względu na charakterystykę wykorzystania CPU:

- zorientowane na wejście-wyjście (*I/O-bound*)
  Wymagają szybkiego przydziału czasu procesora i częstszych przełączeń kontekstu, aby efektywnie obsługiwać operacje wejścia-wyjścia.
- zorientowane na obliczenia (*CPU-bound*)
  Wymagają dłuższego czasu procesora i rzadszych przełączeń kontekstu, aby zmaksymalizować wydajność obliczeniową

Wraz ze wzrostem szybkości CPU procesy mają tendencję do coraz większego ograniczenia przez operacje wejścia-wyjścia.

Cele algorytmów szeregowania w różnych środowiskach

Wszystkie systemy

- **sprawiedliwość** – zapewnienie procesom uczciwego dostępu do CPU
- **równowaga** – dążenie do zajęcia wszystkich części systemu

Systemy interaktywne

- **krótki czas odpowiedzi** (*response time*) – maksymalizacja responsywności procesów interaktywnych
- **proporcjonalność** – spełnianie oczekiwań użytkowników dotyczących czasu wykonania

Systemy wsadowe

- **przepustowość** – maksymalizacja liczby wykonanych zadań na godzinę
- **czas cyklu przetwarzania** – minimalizacja czasu między rozpoczęciem a zakończeniem procesu
- **wykorzystanie CPU** – maksymalizacja jego zajętości

Systemy czasu rzeczywistego

- **dotrzymywanie terminów** – unikanie utraty danych
- **przewidywalność** – unikanie degradacji jakości w systemach multimedialnych

Planowanie przydziału procesora - mechanizmy

- **Algorytmy niewywłaszczające** (*Non-preemptive*)
  - **FCFS** (*First-Come, First-Served*) pierwszy nadszedł, pierwszy obsłużony
  - **SFJ** (*Shortest Job First*) najkrótsze zadanie najpierw
- **Algorytmy wywłaszczające** (*Preemptive*)
  - **RR** (*Round-Robin*) rotacyjne, karuzelowe, cykliczne
  - **SRTF** (*Shortest Remaining Time First*) najpierw najkrótszy pozostały czas
  - **Priorytetowy**: priorytety statyczne i dynamiczne
- **Algorytmy Zaawansowane** stosowane we współczesnych systemach
  - **Kolejki wielopoziomowe** - zestaw kolejek o różnych priorytetach, każdy poziom może mieć inną politykę planowania
  - **Kolejki Wielopoziomowe ze Sprzężeniem Zwrotnym** (MLFQ, *Multi-Level Feedback Queue*) – procesy mogą przemieszczać się między kolejkami w zależności od swojego zachowania
    Przykład: planista Linux (O(1), CFS), planista Windows NT, 10, 11

Szeregowanie bez wywłaszczenia

Algorytm szeregowania **bez wywłaszczania** (*non-preemptive*) wybiera proces do uruchomienia, a następnie pozwala mu działać do czasu zablokowania (np. podczas operacji wejścia-wyjścia) lub do momentu, gdy proces dobrowolnie zwolni CPU.

- prosty w implementacji, małe narzuty związane z przełączaniem kontekstu
- przerwanie zegarowe nie powoduje podjęcia decyzji o szeregowaniu
- wykorzystywany w systemach wsadowych, niewymagających interakcji z użytkownikiem
- może prowadzić do mniejszej responsywności systemu dla procesów interaktywnych
- ryzyko głodzenia procesów o niższym priorytecie

Szeregowanie z wywłaszczeniem

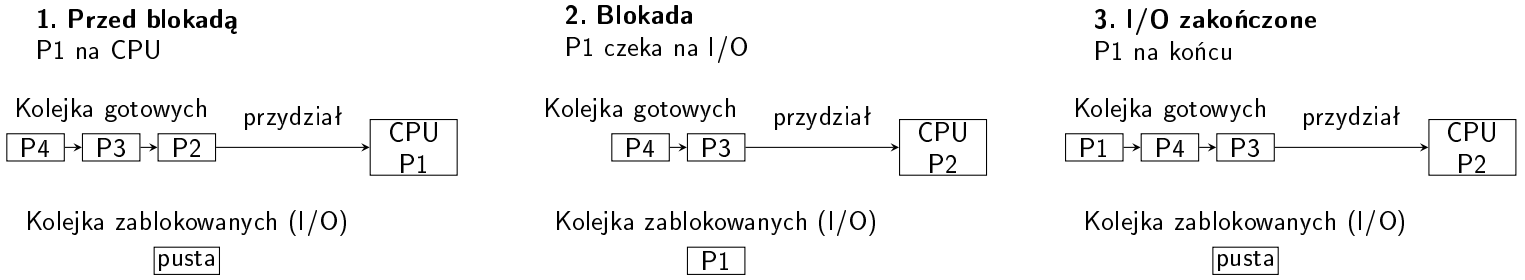
Algorytm szeregowania **z wywłaszczaniem** (*preemptive*) przydziela procesowi CPU na ustalony czas (kwant).

- proces może zostać wywłaszczony, zwykle wskutek przerwania zegarowego, co pozwala planiście ponownie zdecydować o przydziale CPU
- sprawiedliwiej dzieli czas procesora w systemach wielozadaniowych
- krótki kwant sprzyja procesom interaktywnym i poprawia responsywność systemu
- częste przełączenia kontekstu zwiększają narzut, a zbyt krótki kwant może obniżyć wydajność systemu
- dłuższy kwant ogranicza narzut związany z przełączaniem kontekstu i sprzyja procesom obliczeniowym kosztem responsywności
- wymaga dodatkowych mechanizmów synchronizacji, więc jest trudniejszy w implementacji

Pierwszy nadszedł, pierwszy obsłużony (FCFS)

First-Come, First-Served

- pojedyncza kolejka gotowych procesów
- procesy są wykonywane w kolejności, w jakiej trafiają do kolejki gotowych, bez względu na czas trwania czy priorytet (kolejka FIFO)
- gdy proces się blokuje, zwalnia CPU, a planista wybiera kolejny proces z kolejki gotowych
- odblokowany proces trafia na koniec kolejki gotowych



Źródło: John Bell, *CPU Scheduling*, sekcja 5.3.1, University of Illinois Chicago, dostęp 9 listopada 2026 r.

FCFS: wady i zalety

Zalety:

- prosty w implementacji
- sprawiedliwy, każdy proces jest obsługiwany w kolejności zgłoszeń
- minimalne narzuty związane z przełączaniem kontekstu
- przydatny w systemach wsadowych, gdzie zadania są długotrwałe i nie wymagają interakcji

Wady:

- długie procesy mogą nadmiernie blokować CPU, na niekorzyść krótkich zadań, tzw. zjawisko **konwoju** (*convoy effect*)
- nie jest optymalny dla systemów interaktywnych, gdzie responsywność jest kluczowa

Najkrótsze zadanie najpierw (SJF, *Shortest Job First*)

- procesy w kolejce gotowych są uporządkowane według przewidywanego czasu wykonania, od najkrótszego do najdłuższego
- proces o najkrótszym przewidywanym czasie wykonania jest wybierany jako pierwszy do uruchomienia
- SJF minimalizuje średni czas oczekiwania procesu, ponieważ krótsze zadania są obsługiwane szybciej, co zmniejsza opóźnienia w kolejce
- w praktyce trudno jest dokładnie oszacować czas wykonania każdego zadania
- dłuższe procesy mogą być stale opóźniane (głodzenie), jeśli nowe, krótsze zadania ciągle się pojawiają w kolejce
- sprawdza się w systemach, w których czas wykonania procesów jest znany lub można go dokładnie oszacować (systemy wsadowe)

Najkrótsze zadanie najpierw (SJF, *Shortest Job First*)



Figure 2-41. An example of shortest-job-first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

Czas oczekiwania w oryginalnym porządku:
8, 12, 16, 20, średnio 14

Czas oczekiwania w kolejności od najkrótszego zadania:
4, 8, 12, 20, średnio 11

Algorytm **najkrótszego pozostałego czasu** (SRT lub SRTF, *Shortest Remaining Time First*)

- wariant SJF z wywłaszczeniem
- proces o najkrótszym pozostałym czasie wykonania jest wybierany do uruchomienia
- SRT stale monitoruje pozostały czas wykonania procesów w kolejce gotowych
- gdy nowy proces o krótszym czasie wykonania pojawia się w kolejce gotowych, bieżący proces jest wywłaszczany i nowy proces jest uruchamiany
- minimalizuje średni czas oczekiwania procesu

Algorytm **najkrótszego następnego procesu** (SPN, *Shortest Process Next*)

- odmiana SJF wykorzystująca historię uruchomień i zachowanie procesów do szacowania czasu wykonania
- przykładowa estymacja czasu wykonania procesu (*aging*) oparta na średniej wykładniczej:

$$T_{n+1} = aT_n + (1 - a)T_{n-1}$$

  gdzie $0 \leqslant a \leqslant 1$.
- czas T_n oznacza rzeczywisty czas wykonania procesu podczas n-tego uruchomienia

Algorytm rotacyjny (*Round-Robin*)

- procesy gotowe do wykonania są zorganizowane w kolejkę, a każdy proces otrzymuje kwant czasu
- algorytm działa podobnie do FCFS, ale z wywłaszczaniem; zawsze wybierany jest pierwszy proces z kolejki
- gdy kwant czasu dobiegnie końca, proces zostaje wywłaszczony i umieszczony na końcu kolejki
- jeśli proces zakończy się przed upływem kwantu czasu lub zostanie zablokowany, CPU zostaje przydzielone kolejnemu procesowi z kolejki
- kwant czasu wynosi typowo około 20–50 ms; wartość optymalna zależy od charakterystyki obciążenia systemu



Figure 2-42. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Planowanie priorytetowe

- decyzja planisty zależy od **priorytetów** przypisanych do zadań; ważniejsze zadania otrzymują wyższe priorytety i mają pierwszeństwo
- priorytety mogą być przydzielane **statycznie** lub **dynamicznie**
- statyczne priorytety są przypisywane na podstawie właściwości procesu i nie zmieniają się w trakcie jego życia
- priorytety dynamiczne mogą zmieniać się w zależności od zachowania procesu; np. procesy interaktywne, które często czekają na dane od użytkownika, mogą być promowane przez podwyższenie priorytetu, aby zwiększyć responsywność
- procesy można pogrupować według priorytetów, tworząc **klasy priorytetów**. Planowanie między klasami odbywa się zgodnie z priorytetami, a w ramach danej klasy – według wybranego algorytmu szeregowania (np. cyklicznego)

Klasy priorytetów

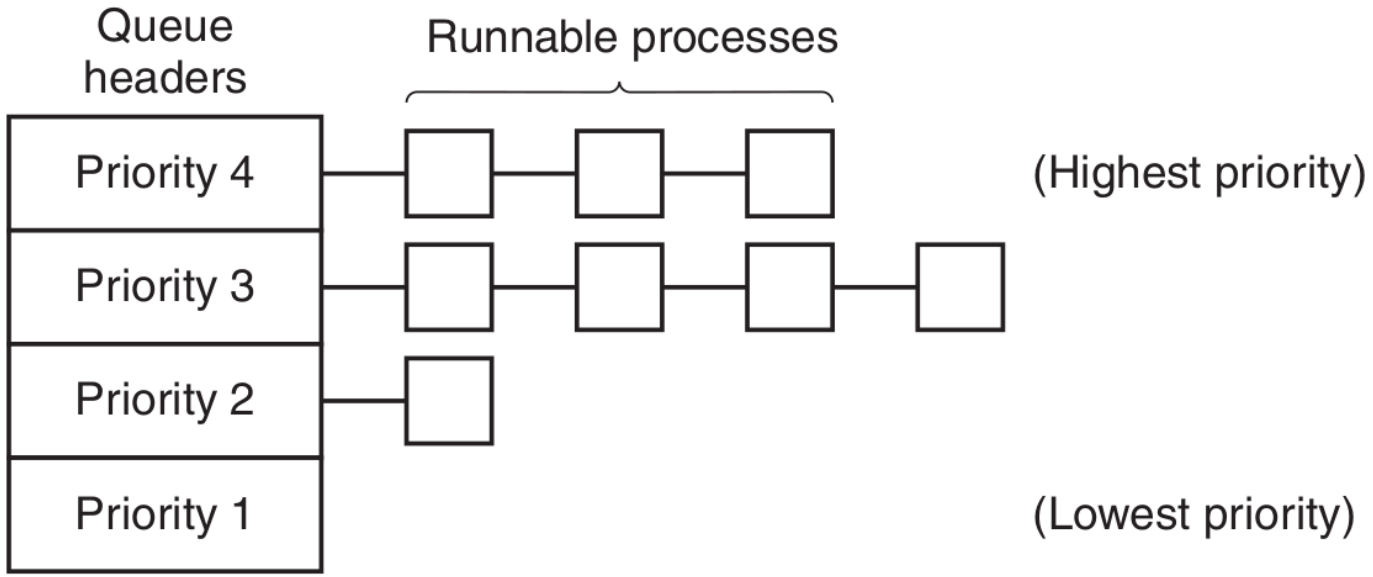


Figure 2-43. A scheduling algorithm with four priority classes.

Algorytm szeregowania w Linuksie

- szeregowane są wątki jądra
- 140 priorytetów: [0, 139], gdzie niższa wartość oznacza wyższy priorytet

Dwie główne klasy zadań:

- zadania czasu rzeczywistego: priorytety statyczne [0, 99]
  polityki planowania: `SCHED_FIFO`, `SCHED_RR`
- zadania normalne: priorytety dynamiczne [100, 139]
  polityki planowania: `SCHED_OTHER`

Algorytmy planowania:

- **planista O(1)** używany od wersji jądra 2.5 (11.2001)
- **planista całkowicie sprawiedliwy** (CFS, *Completely Fair Scheduler*) o złożoności O(log N), od wersji 2.6.23 (10.2007)
- **algorytm EEVDF** (*Earliest Eligible Virtual Deadline First*) zmienił scheduling w klasie `fair` w jądrze 6.6 (10.2023)

Nikita Ishkov, *A complete guide to Linux*

Nazwa nie odnosi się do systemów czasu rzeczywistego i nie oznacza spełniania rygorystycznych ograniczeń czasowych

Priorytety czasu rzeczywistego

- priorytety zadań czasu rzeczywistego mieszczą się w zakresie 0–99 i nie zmieniają się w trakcie wykonywania
- zadania są wykonywane ściśle według malejących priorytetów: najpierw zadania czasu rzeczywistego, a następnie pozostałe

Polityki szeregowania czasu rzeczywistego:

- `SCHED_FIFO` (*First-In, First-Out*) – szeregowanie bez wywłaszczenia, analogiczne do FCFS. Stosowana w przypadku zadań, które muszą być wykonane jak najszybciej i nie mogą być przerwane przez inne procesy o tym samym priorytecie.
- `SCHED_RR` (*Round-Robin*) – szeregowanie rotacyjne (z wywłaszczeniem)

Procesy `SCHED_FIFO` mają zawsze pierwszeństwo przed procesami `SCHED_RR` o tym samym priorytecie.

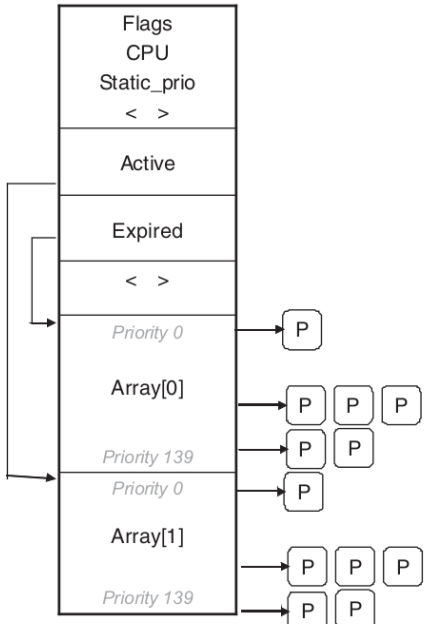
RHEL/CentOS/Fedora ograniczają maksymalne zużycie CPU dla wątków czasu rzeczywistego do 95%; zob. Real Time Scheduler Throttling

Priorytety dynamiczne w planiście O(1)

- klasa `SCHED_OTHER` – standardowe procesy użytkownika o priorytetach dynamicznych z zakresu 100–139
- początkowa wartość priorytetu dynamicznego jest określana na podstawie poziomu uprzejmości (*nice*)
- poziomy uprzejmości od -20 do $+19$ są mapowane na wartości od 100 do 139; domyślna wartość uprzejmości 0 odpowiada priorytetowi 120
- wartość priorytetu procesu zmienia się dynamicznie w granicach ± 5 w zależności od interaktywności zadania wyznaczanego jako stosunek czasu, jaki zadanie spędza w zawieszeniu, do czasu aktywności
- wielkość kwantu czasu procesu zależy od priorytetu dynamicznego; procesy o wyższym priorytecie otrzymują dłuższy kwant czasu
- domyślnie proces otrzymuje kwant 100 ms. Może on zostać zwiększony do 800 ms (100) lub zmniejszony do 5 ms (139).
  Nowy proces dziedziczy kwant po rodzicu.

Planista O(1)

- struktura zarządzania aktywnymi zadaniami (*runqueue*) zawiera dwie tablice o rozmiarze 140: tablicę zadań aktywnych (*active*) i tablicę zadań wygasłych (*expired*)
- każdy element tych tablic wskazuje na początek kolejki zadań o wspólnym priorytecie



(a) Per CPU runqueue in the Linux O(1) scheduler

- planista wybiera pierwsze zadanie z listy aktywnych o najwyższym priorytecie
- po wyczerpaniu kwantu czasu zadanie jest umieszczane na liście wygasłych (możliwa jest zmiana priorytetu)
- jeśli zadanie zostanie zablokowane przed upływem kwantu czasu, powraca do listy aktywnych z odpowiednio skróconym kwantem
- gdy wszystkie listy aktywnych zadań się wyczerpią, następuje zamiana tablic i zadania wygasłe stają się aktywne
- wybór zadania odbywa się w stałym czasie $O(1)$, niezależnie od liczby zadań
- algorytm gwarantuje, że wszystkie zadania, nawet o niskim priorytecie, mają szansę wykonania
- główną wadą algorytmu jest skomplikowana heurystyka ustalania priorytetów dynamicznych, promująca zadania interaktywne, kosztem zadań obliczeniowych

Co oznacza PRI?

proces/wątek	jądro	"top/htop"	"ps -eo pri"	"ps -eo rtprio"	"ps -el"	"nice"
migration	99	RT	139	99	-40	-
kworker	100	0	39	-	60	-20
bash	120	20	19	-	80	0
khugepaged	139	39	0	-	99	19

	priorytet wg jądra $\geqslant 100$	priorytet wg jądra < 100
top/htop	PRI='priorytet wg jądra' - 100	PRI=RT
ps -eo pri	PRI=139 - 'priorytet wg jądra'	PRI='priorytet wg jądra' + 40
ps -el	PRI='priorytet wg jądra' - 40	PRI='priorytet wg jądra' - 139

Obecne klasy szeregowania w jądrze Linuksa

Klasy zadań objęte wspólną polityką przydziału CPU w kolejności priorytetu, od najwyższego do najniższego:

1. wstrzymania (*stop*): kluczowe wątki jądra, które musza być wykonane natychmiast
2. terminowa (*deadline*): `SCHED_DEADLINE` dla zadań o ścisłych wymaganiach czasowych
3. czasu rzeczywistego (*real time*): `SCHED_RR`, `SCHED_FIFO`
4. sprawiedliwa (*fair*): `SCHED_NORMAL` (dawniej `SCHED_OTHER`); obejmuje większość procesów użytkownika zarządzanych algorytmem EEVDF (wcześniej CFS)
5. wsadowa (*batch*): `SCHED_BATCH` dla nieinteraktywnych zadań obciążających CPU o niskim priorytecie
6. bezczynna (*idle*): `SCHED_IDLE` dla procesów o bardzo niskim priorytecie (niższym niż +19), które mogą być wykonywane tylko wtedy, gdy system jest bezczynny

Przykład: polityka szeregowania procesów w Linuksie

```
$ ps -e -o s,pid,policy,pri,ni,rtprio,command
S     PID POL PRI  NI RTPRIO COMMAND
S       1 TS   19   0      - /sbin/init splash
S       2 TS   19   0      - [kthreadd]
I       4 TS   39 -20      - [kworker/R-rcu_g]
...
S      19 FF  139   -     99 [migration/0]
S    4565 B     4  15      0 /usr/libexec/packagekitd
S    5670 TS    7  12      - /usr/bin/python3 /usr/lib/pop-transition/service.py
R   13056 IDL   0   -      0 /usr/libexec/tracker-miner-fs-3
S   30080 TS   13   6      - /usr/bin/python3 /usr/bin/gnome-terminal --wait
R  464202 TS   19   0      - ps -e -o s,pid,policy,pri,ni,rtprio,command
```

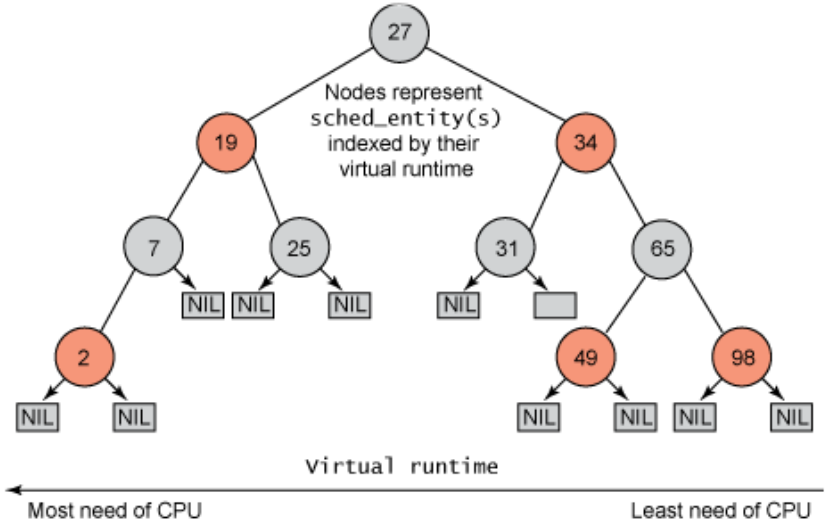
Polityki szeregowania: `TS` (SCHED_NORMAL), `FF` (SCHED_FIFO), `RR` (SCHED_RR), `B` (SCHED_BATCH), `IDL` (SCHED_IDLE), `DL` (SCHED_DEADLINE)

Polecenie `chrt` ustawia lub odczytuje atrybuty szeregowania

```
$ chrt -p 19
pid 19's current scheduling policy: SCHED_FIFO
pid 19's current scheduling priority: 99
```

Planista całkowicie sprawiedliwy CFS

- struktura zarządzania zadaniami (`runqueue`) jest zorganizowana w postaci drzewa czerwono-czarnego, w którym zadania gotowe do wykonania są uporządkowane według **czasu wirtualnego** (wartość `vruntime`)
- planista wybiera do wykonania zadanie o najmniejszym czasie wirtualnym (skrajny lewy węzeł)



http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

Planista CFS

- **Czas wirtualny** to rzeczywisty czas wykonania podzielony przez wagę zadania
- waga zadania zależy od priorytetu (`nice`), niższa uprzejmość oznacza wyższą wagę
- zadania o wyższej wadze (mniej uprzejme) mają wirtualny czas płynący wolniej, co pozwala im na dłuższe uruchomienie
- dla zadań o niskim priorytecie (o niskiej wadze) czas płynie szybciej, dlatego są one szybciej wywłaszczane na rzecz zadań o wyższym priorytecie
- planista okresowo aktualizuje wirtualny czas wykonywanego zadania, porównując go z czasem kolejnego zadania w drzewie
- jeżeli w drzewie znajduje się węzeł o niższym czasie wirtualnym, następuje wywłaszczenie zadania i umieszczenie go ponownie w drzewie we właściwym miejscu (według nowego czasu wirtualnego)

Algorytm EEVDF (Earliest Eligible Virtual Deadline First)

Najwcześniejszy kwalifikowany termin wykonania w pierwszej kolejności

- planista wyznacza **czas wirtualny** dla każdego zadania w ten sam sposób jak w CFS
- średni czas wirtualny wszystkich procesów wyznacza **uczciwą linię** (*fair line*) V, która reprezentuje idealne, sprawiedliwe rozdzielenie CPU
- wirtualne **opóźnienie** (*lag*) zadania to różnica między jego czasem wirtualnym a uczciwą linią

$$lag = vruntime - V$$

- **Kwalifikowalność** (*eligibility*) – zadanie jest kwalifikowane do uruchomienia tylko wtedy, gdy jego czas wirtualny nie przekracza uczciwej linii, czyli gdy

$$lag \leqslant 0 \qquad vruntime \leqslant V$$

- procesy z dodatnim opóźnieniem (które już wiele dostały) są wstrzymywane aż do wyrównania

- **Wirtualny termin wykonania** (*virtual deadline*) – każdy proces otrzymuje kawałek czasu CPU (*slice*), a z niego obliczany jest wirtualny termin wykonania:

$$deadline = vruntime + (slice/weight)$$

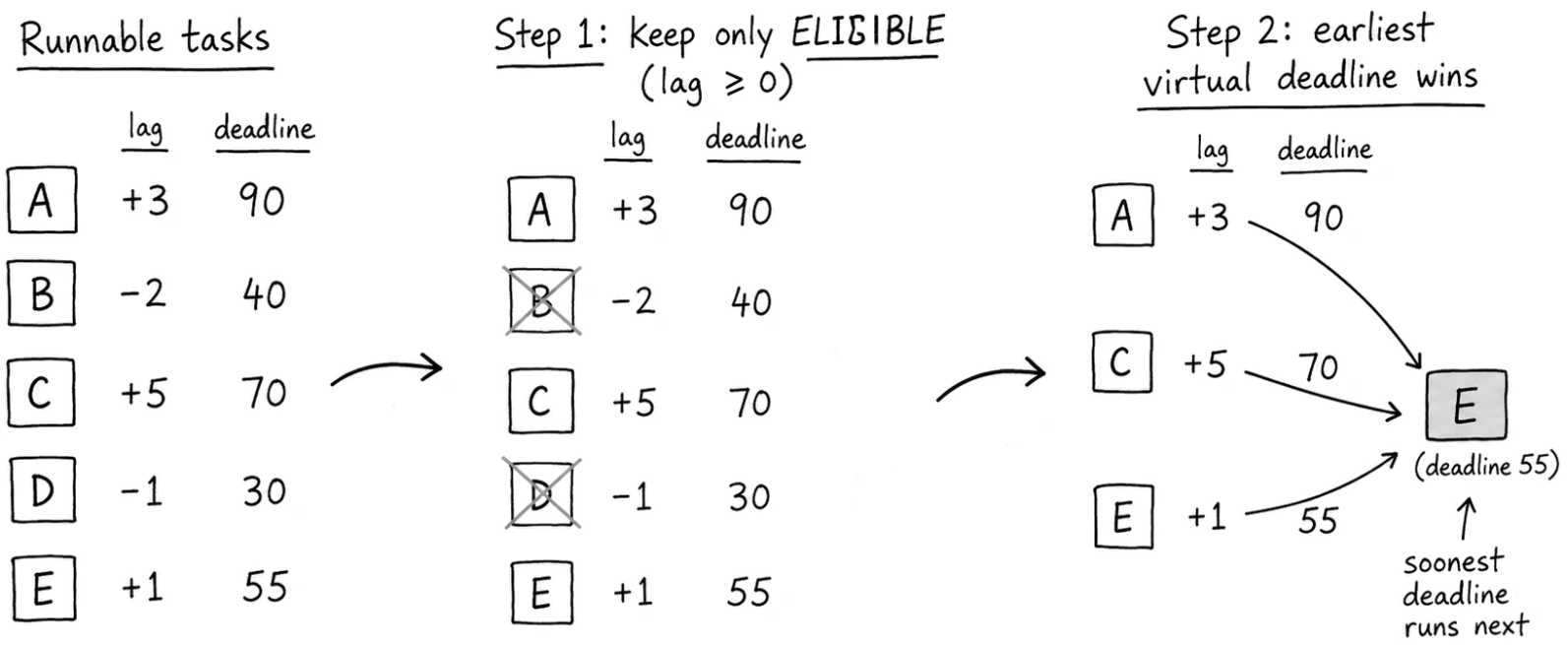
  - procesy o krótszym żądanym slice otrzymują wcześniejszy termin
  - procesy o wyższej wadze mają wcześniejszy termin dla tego samego slice

- Algorytm wyboru: EEVDF wybiera spośród **kwalifikowanych** procesów ten z **najwcześniejszym terminem wykonania**

- `runqueue` zorganizowana jest w drzewie czerwono-czarnym sortowanym po terminie wykonania, co pozwala na wybór w czasie $O(\log n)$

- drzewo jest wzmocnione informacją o minimalnym `vruntime` w każdym poddrzewie, co pozwala na efektywne filtrowanie niekwalifikowanych procesów

Algorytm wyboru EEVDF



Porównanie EEVDF z CFS

- CFS wybierał proces z najmniejszym `vruntime`, co mogło prowadzić do niestabilnych opóźnień (*jitter*) dla procesów interaktywnych
- EEVDF gwarantuje bardziej deterministyczne opóźnienia poprzez koncepcję wirtualnego terminu wykonania
- EEVDF lepiej obsługuje procesy o różnych żądanych slice'ach, preferując krótkie żądania (bardziej interaktywne procesy)
- zarówno CFS jak i EEVDF zapewniają sprawiedliwość poprzez wagę i kwalifikowalność

W algorytmie EEVDF waga zapewnia priorytet, kwalifikowalność zapewnia sprawiedliwość, a wirtualny termin wykonania kontroluje opóźnienie.

Pytania sprawdzające

- Jakie jest zadanie planisty krótkoterminowego?
- Jakie cele realizują algorytmy planowania procesów w systemach operacyjnych ogólnego przeznaczenia?
- Jak wpływa wielkość kwantu czasu na efektywność działania algorytmu rotacyjnego (*round robin*)?
- Jaka jest różnica między algorytmem planowania FCFS i algorytmem rotacyjnym (*round robin*)?
- Jaka jest domyślna wartość priorytetu statycznego procesu uruchomionego przez użytkownika w systemie Linux?