Szeregowanie zadań w Linux Kernel 2.6

Daniel Górski
Przemysław Jakubowski

Plan prezentacji

- Szeregowanie procesów
  - Szeregowanie
  - Cele szeregowania
  - Scheduler 2.6
  - Struktury danych używane w 2.6
  - Multiprocesorowość
  - Tuning
  - 2.4 versus 2.6

Wstęp do szeregowania

- Sprawiedliwość
  - zapobieganie zagłodzeniu
- Kosztowność
  - czas na zmienę kontekstu
- Dokonywanie wyboru

Scheduler

Cele:

- Duża inteakcyjność podczas obciążenia systemu
- Sprawiedliwość
- Priorytety

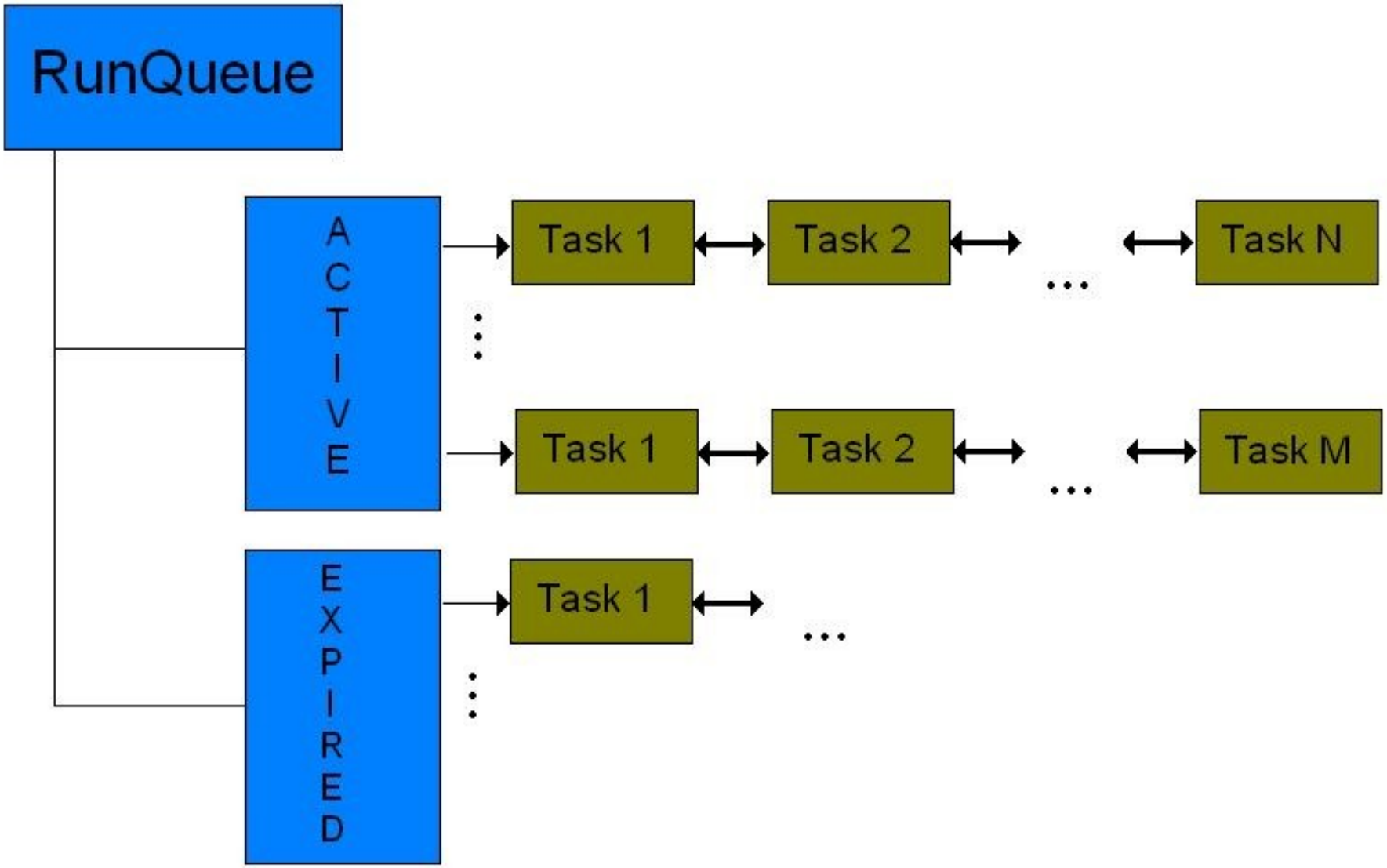
Cele ...

- Procesy czasu rzeczywistego
- Pełne wykorzystanie multiprocesorowości
  - brak globalnych struktur
- Szeregowanie w czasie O(1)

Projekt schedulera 2.6

- 140 poziomów priorytetu
  - mniejsza wartość – większy priorytet
- Dwie tablice priorytetów dla każdego procesora
  - tablica aktywnych: te którym zostały jakieś kwanty czasu
  - tablica zużytych
  - są dostępne poprzez wskaźniki
- Wskaźniki są zamieniane przy zmianie epoki

Kolejka dla jednego procesora



Algorytm O(1)

- Wybierany jest największy priorytet z co najmniej jednym zadaniem w kolejce
  - Pierwsza tablica w bitmapie zero – jedynkowej (operacja bitowa – szybka)
- Pierwszy proces z wybranej kolejki jest uruchamiany
  - Czas stały
- Niezależnie od ilości procesów, wybór nowego do wykonywania zajmuje czas stały
- Algorytm deterministyczny

O(1)...

- W momencie gdy proces kończy działanie jest przenoszony do kolejki zużytych (choć nie zawsze)
- Jest obliczany dla niego nowy priorytet dynamiczny i nowy kwant czasu
- Zamiast n operacji przy zmianie epoki, mamy jedno wyliczanie przy każdej zmianie kontekstu

Strategia szeregowania w 2.6

- 140 priorytetów
- 0-99 priorytety procesów czasu rzeczywistego
- 100-140 priorytety procesów użytkownika
- Trzy różne strategie szeregowania
  - Jedna dla normalnych zadań
  - Dwie dla procesów czasu rzeczywistego

Strategia szeregowania zwykłych procesów

- Każdy proces ma wartość “Nice” (static_priority)
- PRIO = MAX_RT_PRIO + NICE + 20 - bonus
- Procesy są wykonywane od najwyższych priorytetów (czyli najmniejsze wartości PRIO)

Gdy wszystkie wykorzystają kwanty czasu, rozpoczyna się nowa epoka (tylko zmiana wskaźników)

Interakcyjność

- Priorytet dynamiczny procesu zależy od jego inteakcyjności
- Procesy korzystające głównie z procesora [+5]
- Procesy interakcyjne [-5]
  - często korzystające z I/O
- - często korzystające ze sleep
- Bonus = CURRENT_BONUS(p) – MAX_BONUS/2
- CURRENT_BONUS(p) = NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS / MAX_SLEEP_AVG)
- Nie dotyczy procesów czasu rzeczywistego !!!

Interakcyjność ...

- interactive_credit
- Rośnie gry proces śpi, maleje gdy proces korzysta z CPU
- Jeśli proces często zasypia na I/O, a czasami korzysta więcej z CPU dostaje mniejszą karę
- Jeśli proces często korzysta z CPU, a czasami zaśnie na dłużej dostanie mniejszy bonus

Ponowne umieszczanie w kolejce aktywnych

- Aby uniknąć przestojów procesy interakcyjne mogą być włożone ponownie do kolejki aktywnych po wykorzystaniu kwantu czasu
- Tylko w przypadku gdy zadania z kolejki zużytych działały ostatnio (best_expired_prio, STARVATION_LIMIT)
  - nie można ich zagłodzić
- Decyzja o ponownym włożeniu zależy od priorytetu

Obliczanie kwantów czasu

- Długość kwantu czasu zależy od priorytetu statycznego

BASE_TIMESLICE(p) = MIN_TIMESLICE +
(MAX_TIMESLICE – MIN_TIMESLICE)*
(MAX_PRIO-1 – (p->static_prio) / (MAX_USER_PRIO-1))

- Proces może jednak zużyc maksymalnie TIMESLICE_GRANULARITY ms czasu

Jak ma się fork() do priorytetów

- Statyczny priorytet pozostaje bez zmian
- Bierzący kwant czasu ojca jest dzielony na pół między jego i syna
- Zmniejszenie sleep_avg rodzica i dziecka
  - w ten sposób rodzic i dziecko bedą miały mniejszy priorytet dynamiczny

Strategie dla procesów czasu rzeczywistego

- SCHED_FIFO
  - proces dobrowolnie zrzeka się procesora
  - Priorytety są stałe
- SCHED_RR
  - Przydział kwantu czasu
  - Proces działa aż wykorzysta go
  - Po wykorzystaniu go, ma przyznawany nowy i jest ponownie umieszczany w kolejce aktywnych!
- Obydwie strategie mogą być niesprawiedliwe
  - możliwe zagłodzenie

Struktura Runqueue

- spinlock_t lock
  - tylko jeden proces może modyfikować kolejkę w danym czasie
- unsigned long nr_running
  - liczba gotowych procesów w kolejce
- unsigned long long nr_switches
  - liczba zmian kontekstu od momentu powstania kolejki (nigdzie nie używane)
- atomic_t nr_iowait
  - liczba procesów czekających na I/O

Runqueue ...

- unsigned long nr_uninterruptible
  - liczba zadań w kolejce, które nieobsługują przerwań
- unsigned long expired_timestamp
  - czas ostatniej zmiany epoki
- int best_expired_prio
  - używane aby nie głodzić procesów

Runqueue ...

- task_t *curr
  - wskaźnik do aktualnie działającego procesu
- task_t *idle
  - wskaźnik do procesu który działa wtedy, gdy nic innego nie działa
- prio_array_t *active
- prio_array_t *expired
  - wskaźniki do tabel

Struktura prio_array

- unsigned int nr_active
  - liczba gotowych zadań w kolejce
- unsigned long bitmap[BITMAP_SIZE]
  - do szukania najwyższego priorytetu
- struct list_head quene[MAX_PRIO]
  - tablica list procesów o danych priorytetach

SMP

- Wiele procesorów
- Posiadają wspólną pamięć
- Gdy któryś jest bardziej obciążony od innych następuje równoważenie ilości zadań

Równoważenie

- Co 1ms, gdy procesor jest bezczynny, co 200ms gdy pracuje wywoływana jest load_balance()
- Szuka najbardziej obciążonego procesora
- Jeśli różnica w ilości zadań mniejsza niż 25% - koniec

Równoważenie ...

- Wybiera kolejkę z której będzie pobierał zadania (preferowana jest zużytych)
- Bierze listę zadań o najniższym priorytecie
- Sprawdza czy zadanie może zostać ściągnięte na dany procesor (mapa bitowa, cache)
- Powtarza dopóki nie są zrównoważone

NUMA

- Komputery podzielone są na węzły, w ramach węzła procesory dzielą pamięć i inne zasoby
- Mamy głównego schedulera i każdy węzeł ma swojego schedulera
- Scheduler węzła zajmuje się podziałem zadań wewnątrz węzła
- Scheduler główny przenosi zadania między węzłami (kosztowne) tylko gdy węzeł jest przeciążony

Tuning schedulera 2.6

- Aby wprowadzić zmiany do schedulera konieczna jest rekomplilacja jądra
- Co można zmienić:
  - MIN_TIMESLICE, MAX_TIMESLICE – przedział wielkości kwantu czasu, zwiększa się ogólna wydajność systemu, kosztem czasu reakcji

Tuning...

- PRIO_BONUS_RATIO
  - zakres bonusu jaki procesy dostają w zależności od ich interakcyjności (w % promienia nice() domyślnie 25%).
  - zwiększając tą wartość zmiejszamy znaczenie priorytetu statycznego nice().

Tuning...

- MAX_SLEEP_AVG – ograniczenie na parametr SLEEP_AVG
  - zwiększenie tej wartości bardziej “wyrówna” rozkład procesora pomiędzy interakcyjne i nieinterakcyjne

Tuning...

- STARVATION_LIMIT – jeśli jakiś proces z kolejki zużytych czeka tyle czasu, nie jest możliwe ponowne wkładanie procesów inteakcyjnych do kolejki aktywnych
  - zwiększenie wspomaga procesy inteakcyjne

Możliwe zmiany

- Możliwość dynamicznej zmiany ustawień schedulera.
- Główny scheduler dzieli czas między użytkowników, użytkownik wybiera rodzaj schedulera dla własnych procesów

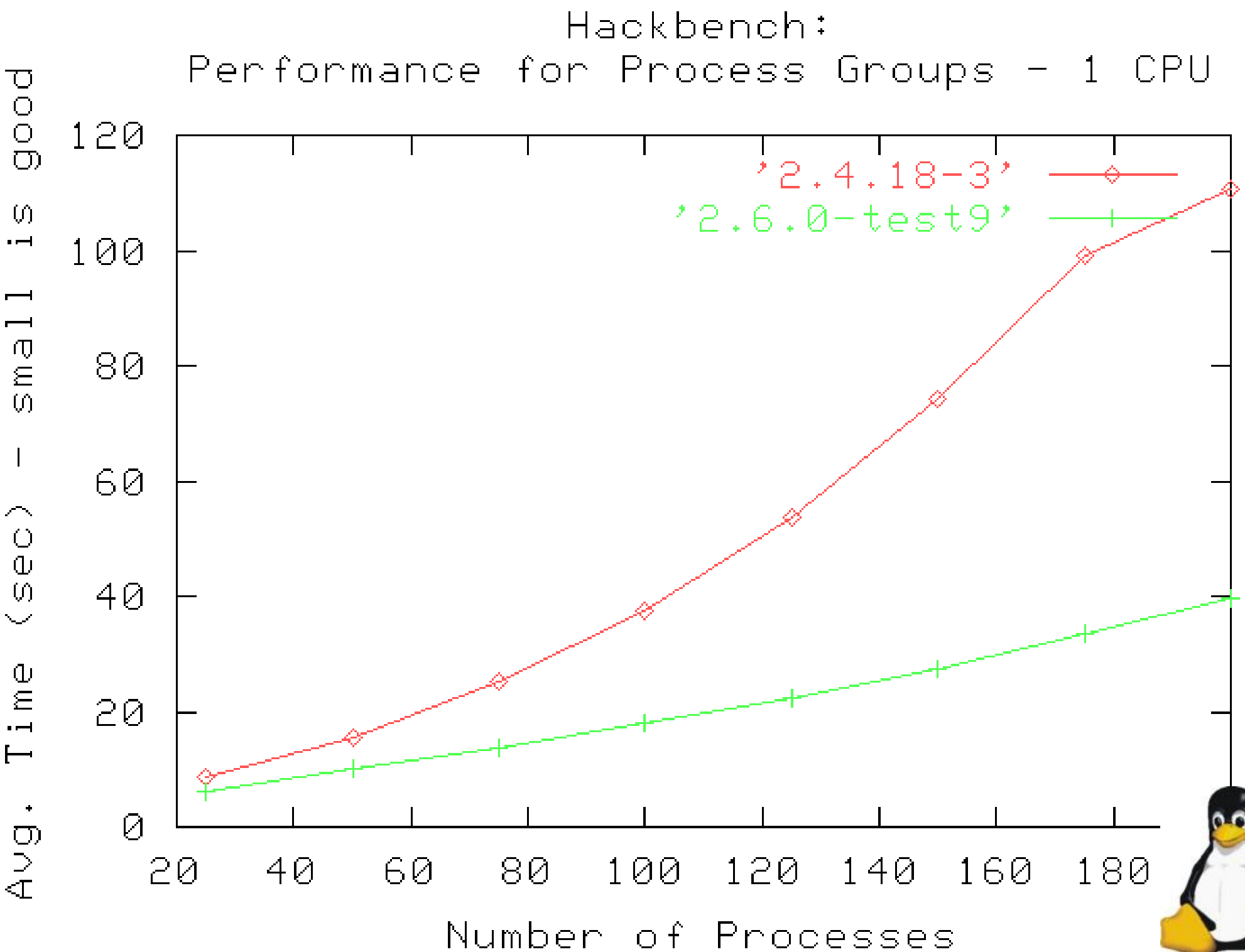
2.4 versus 2.6

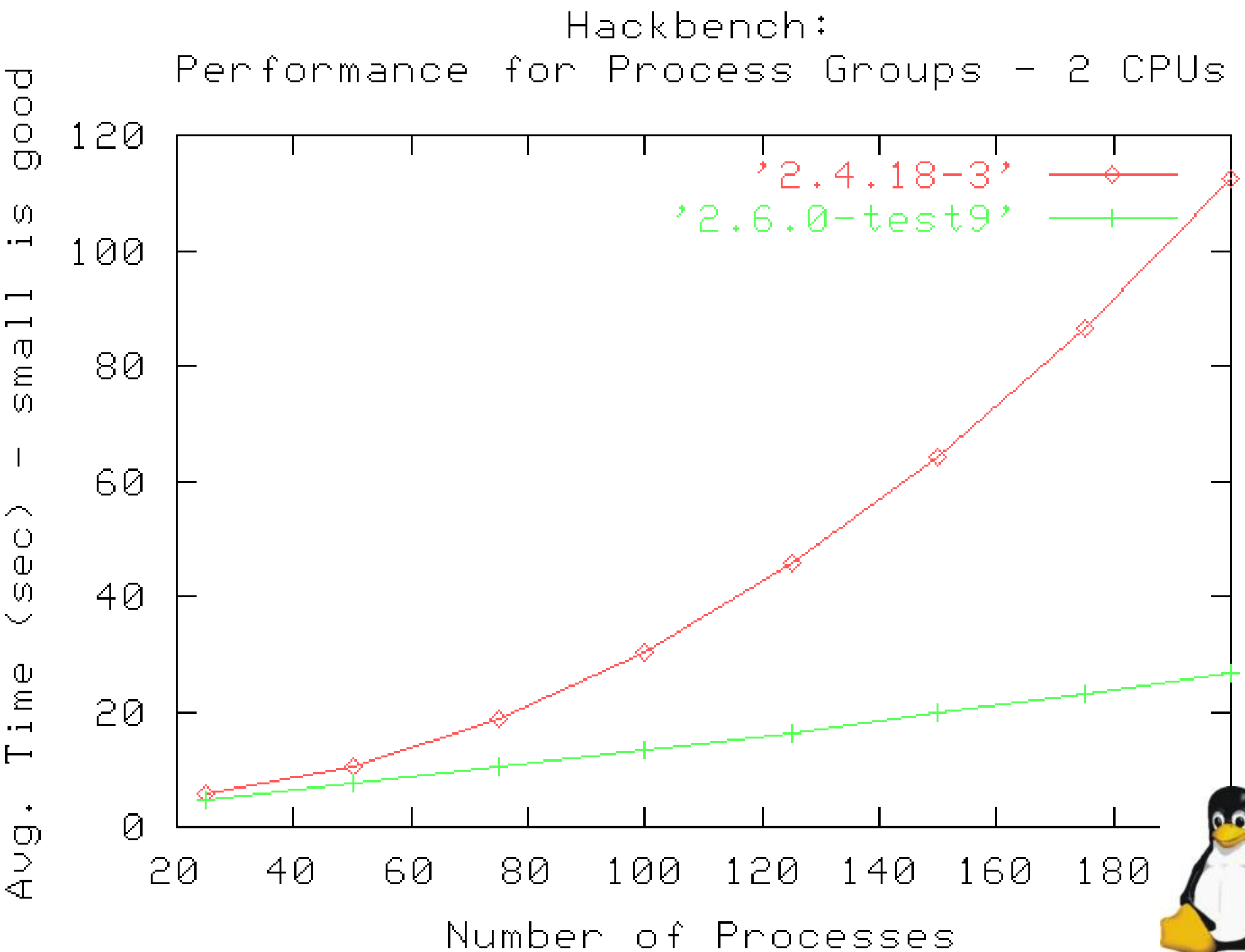
Scheduler w jądrze 2.4:

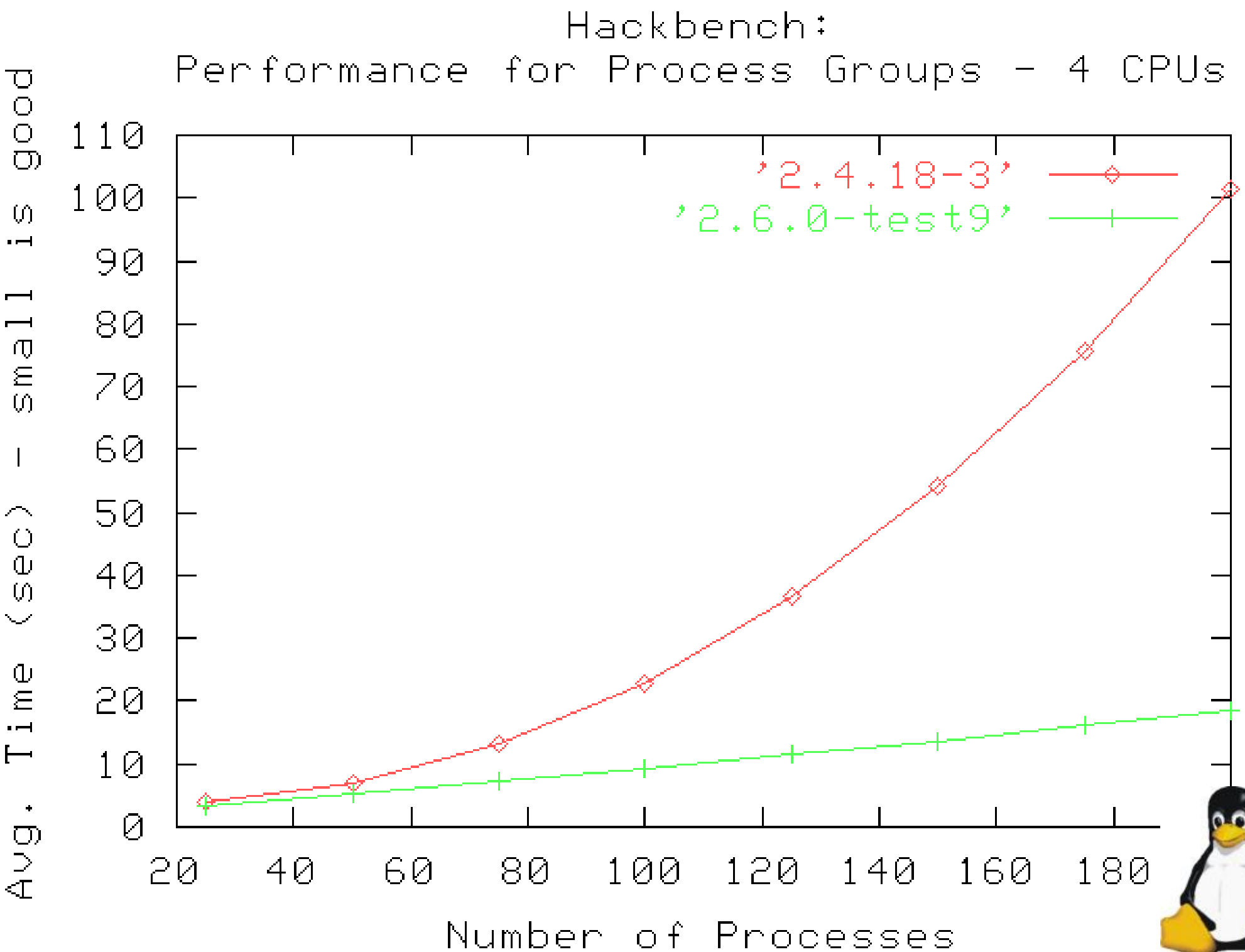
- Globalna kolejka
  - wszystkie procesory muszą czekać aż inne skończą wybór
- Algorytm O(n)
  - przegląda całą listę, aby wybrać następne zadanie
- Zajmuje dużo czasu przy obciążonym systemie
- Nie można wywłaszczać RT

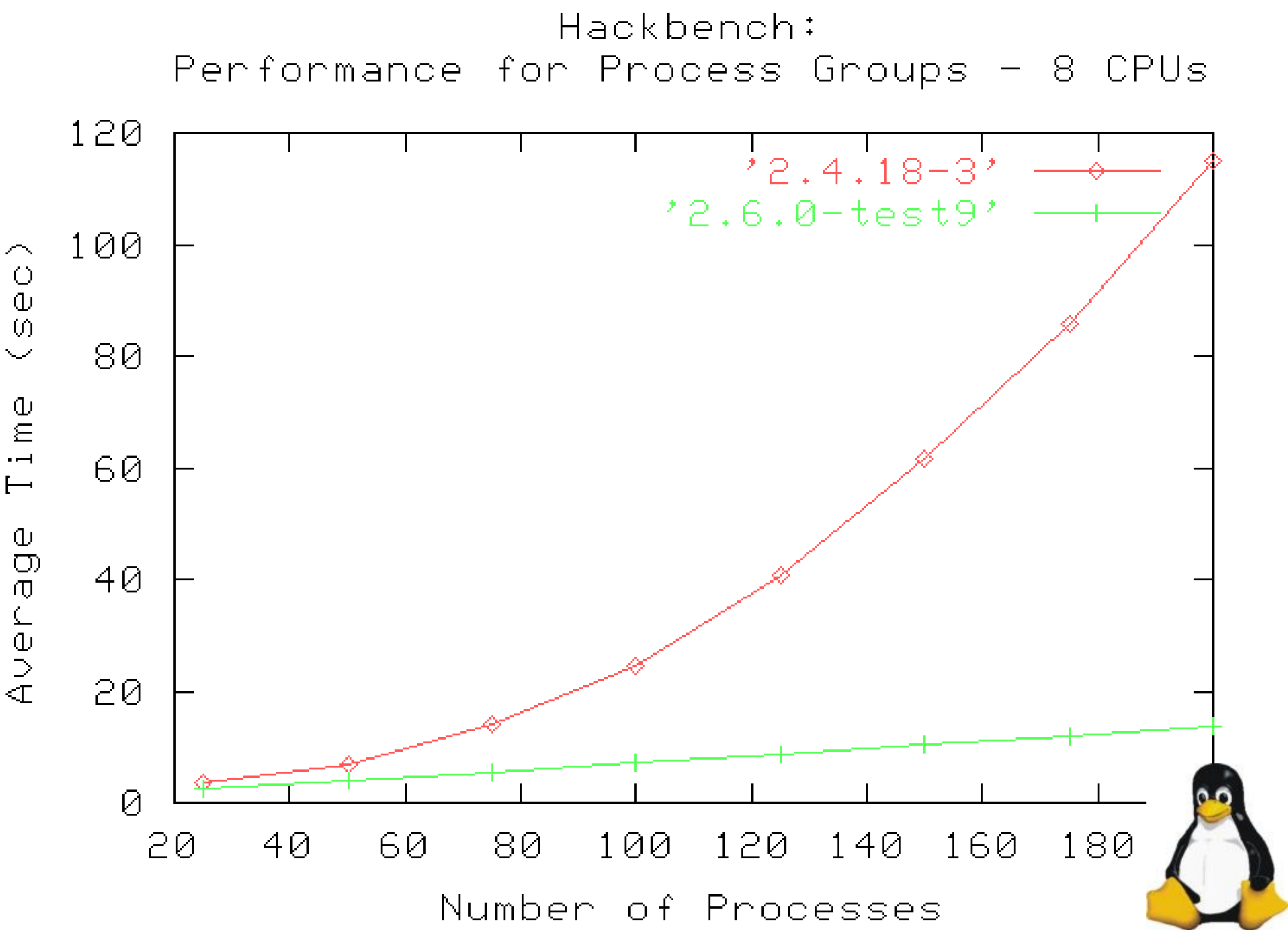
2.4 versus 2.6

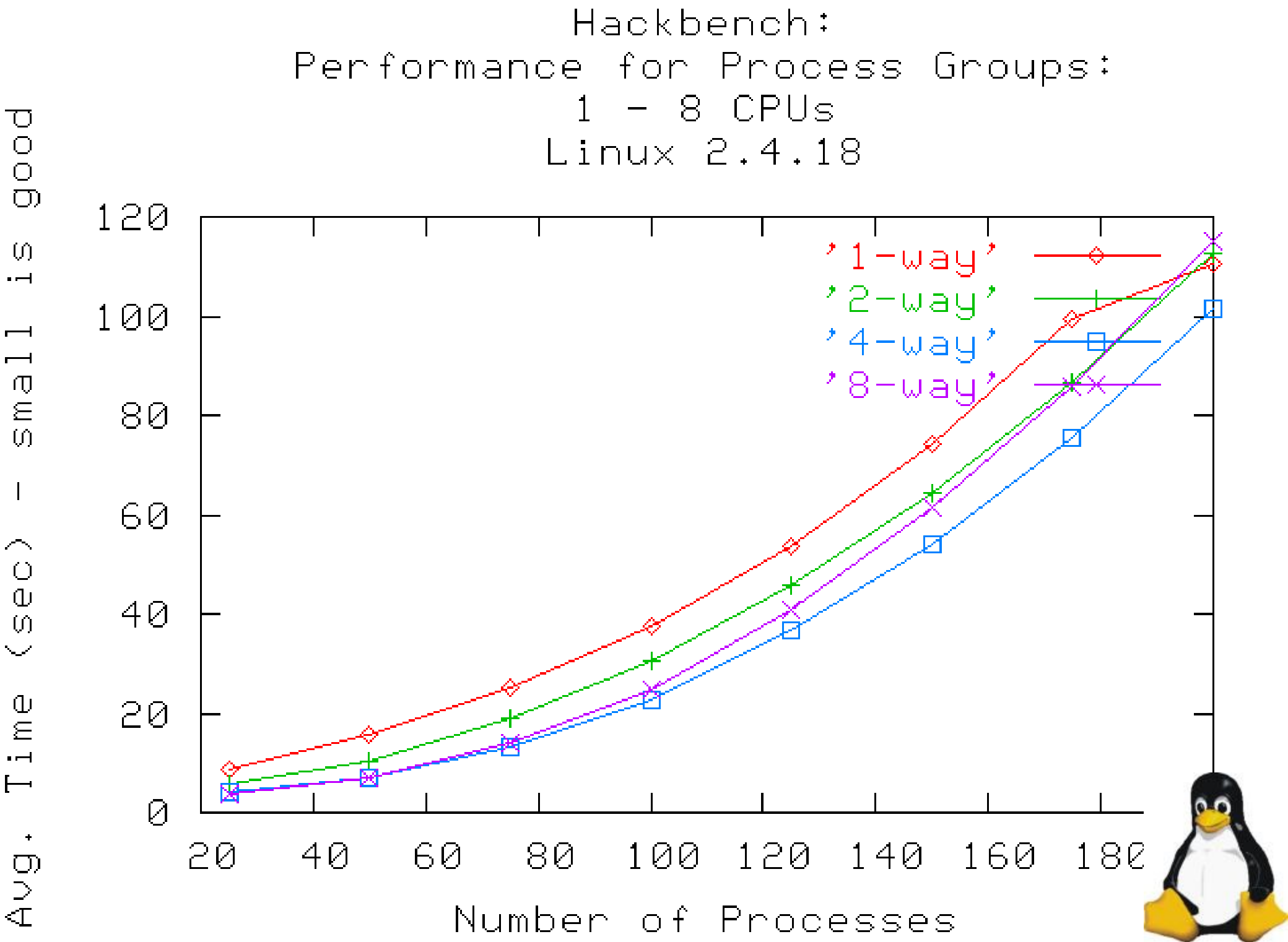
- Jak test był wykonany:
  - uruchomiona wielokrotność 25 par klient/serwer procesów
  - każdy klient wysyłał do każdego serwera 100 komunikatów, serwery nasłuchiwały
  - każdy test wykonywany był kilka razy, pod uwagę był brany średni czas wykonywania testu

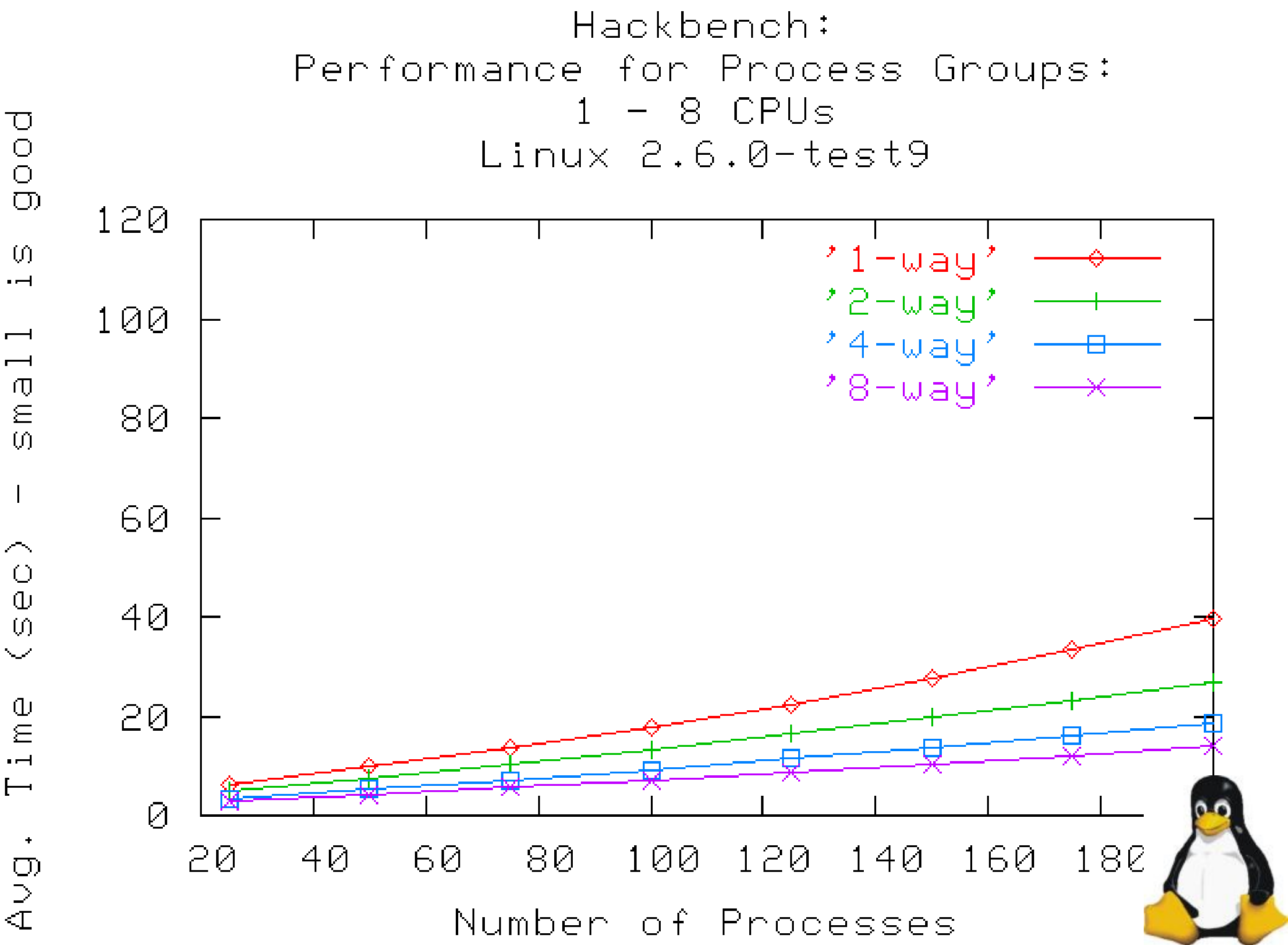

Hackbench:
Performance for Process Groups - 1 CPU
'2.4.18-3'
'2.6.0-test9'
Avg. Time (sec) - small is good
Number of Processes
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180

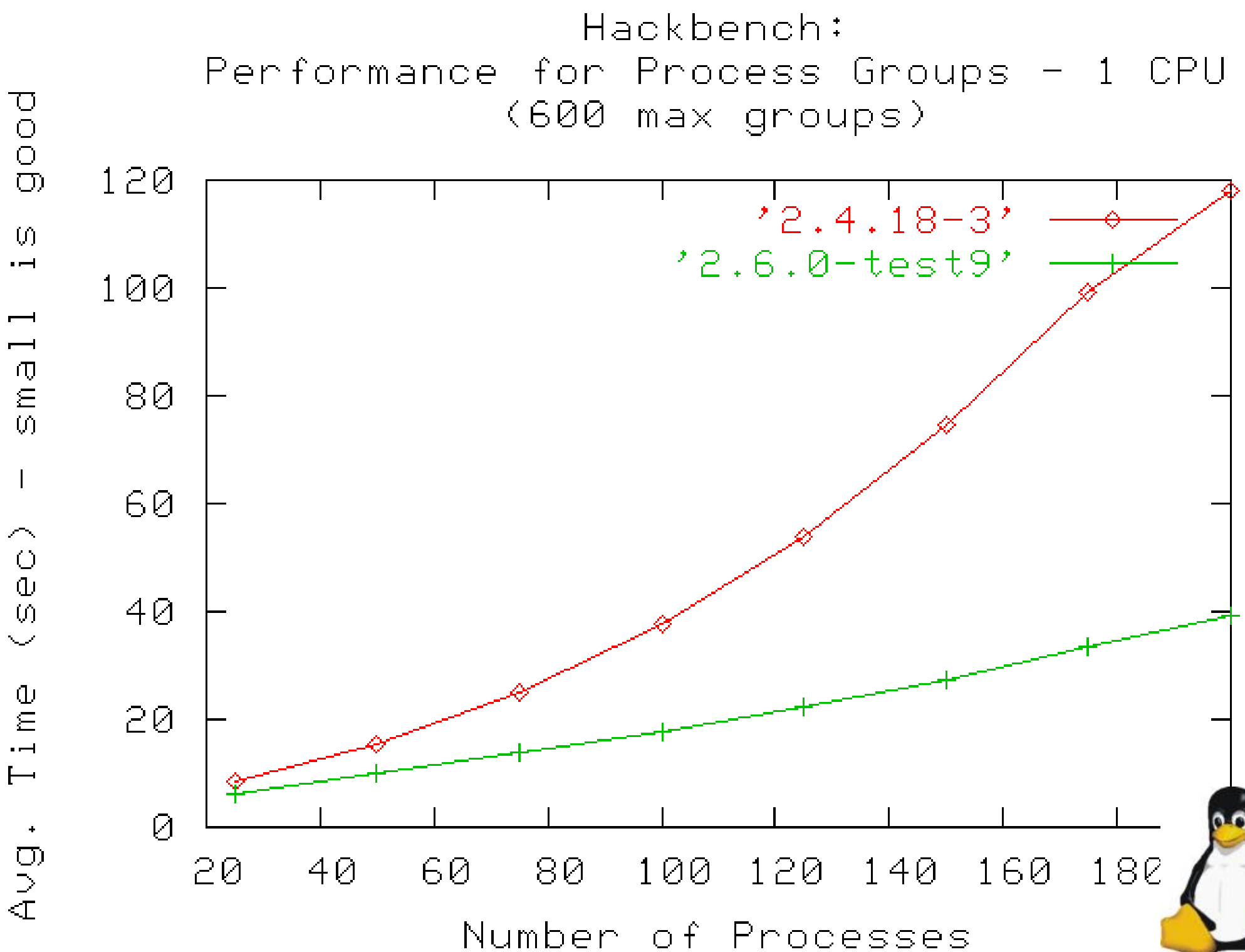

Hackbench:
Performance for Process Groups - 2 CPUs
'2.4.18-3'
'2.6.0-test9'
Avg. Time (sec) - small is good
Number of Processes
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180

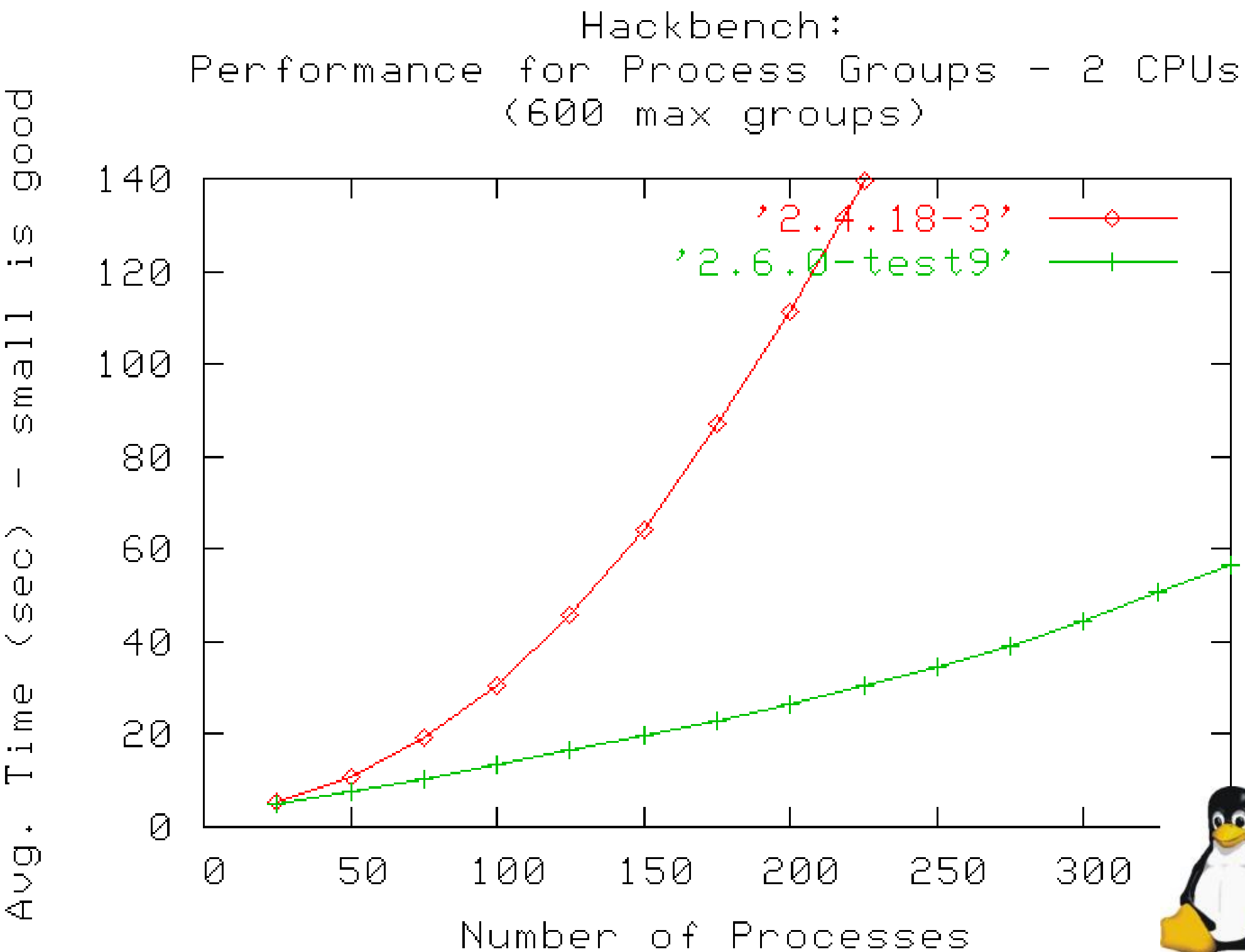

Hackbench:
Performance for Process Groups - 4 CPUs
'2.4.18-3'
'2.6.0-test9'
Avg. Time (sec) - small is good
Number of Processes
0
10
20
30
40
50
60
70
80
90
100
110
20
40
60
80
100
120
140
160
180

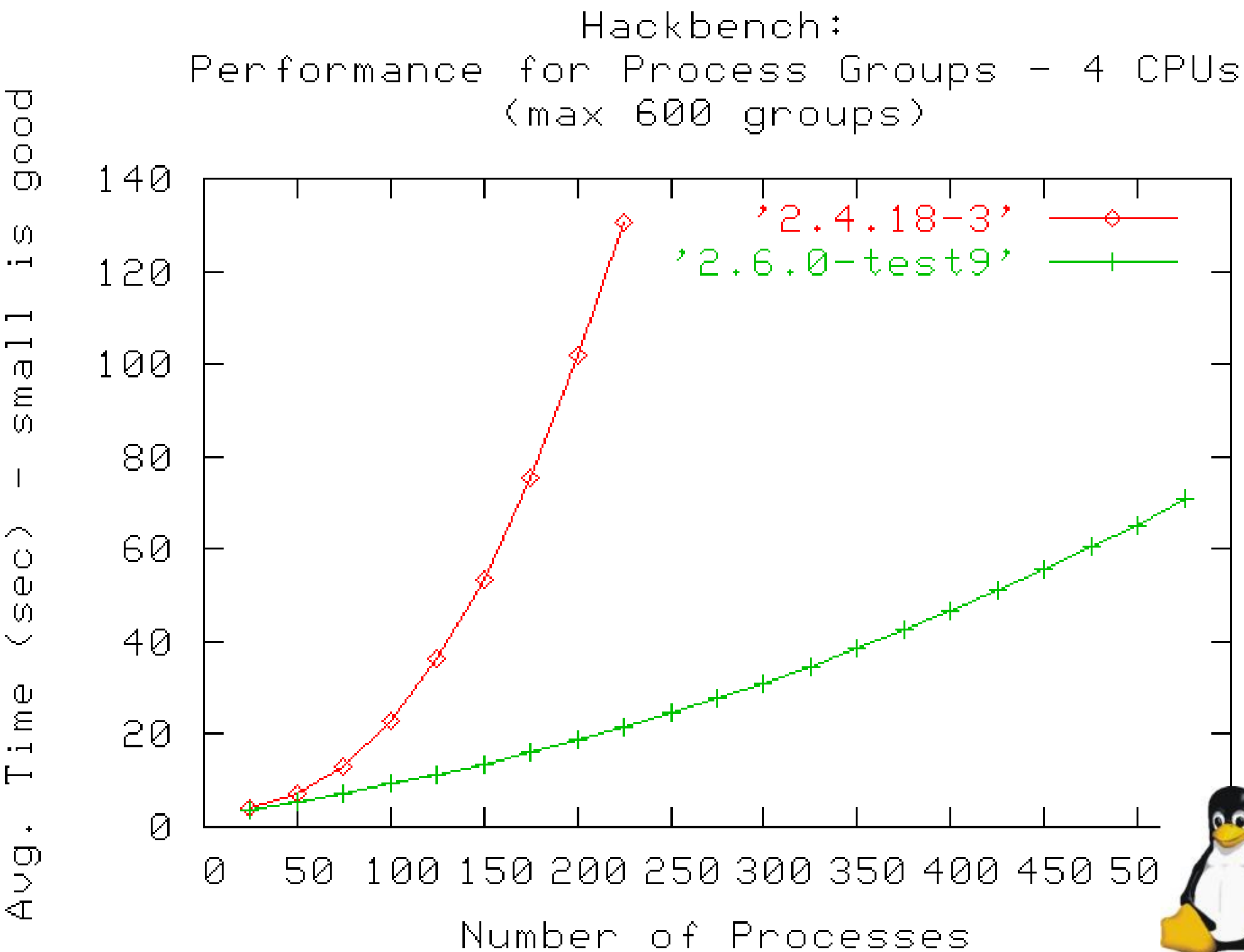

Hackbench:
Performance for Process Groups - 8 CPUs
'2.4.18-3'
'2.6.0-test9'
Average Time (sec)
Number of Processes
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180


Hackbench:
Performance for Process Groups:
1 - 8 CPUs
Linux 2.4.18
Avg. Time (sec) - small is good
Number of Processes
'1-way'
'2-way'
'4-way'
'8-way'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180


Hackbench:
Performance for Process Groups:
1 - 8 CPUs
Linux 2.6.0-test9
Avg. Time (sec) - small is good
Number of Processes
'1-way'
'2-way'
'4-way'
'8-way'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180


Hackbench:
Performance for Process Groups - 1 CPU
(600 max groups)
'2.4.18-3'
'2.6.0-test9'
Avg. Time (sec) - small is good
Number of Processes
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180


Hackbench:
Performance for Process Groups - 2 CPUs
(600 max groups)
Avg. Time (sec) - small is good
Number of Processes
'2.4.18-3'
'2.6.0-test9'
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300


Hackbench:
Performance for Process Groups - 4 CPUs
(max 600 groups)
Avg. Time (sec) - small is good
Number of Processes
'2.4.18-3'
'2.6.0-test9'
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300
350
400
450
50