Управління обчислювальними процесами (3)

01. Оптимізація планування паралельного виконання обчислювальних процесів (ОП)
02. Прив’язка ОП до процесора (Processor affinity)
03. Планування паралельного виконання обчислювальних процесів в ОС UNIX
04. Планування паралельного виконання обчислювальних процесів в ОС Linux

01.1. Оптимізація планування паралельного виконання ОП

Параметри планування
(критерії ефективності роботи планувальника):

1. Пропускна здатність (throughput) → кількість обчислювальних процесів (обчислень) виконаних за одиницю часу; (→ max)

2. Затримка (→ min):
2.1. Загальний час виконання процесу з моменту старта до моменту завершення (turnaround time);
2.2. Час відгуку на запит (наприклад, реакція на дію користувача в GUI) (response time) → average response time vs. variance in the response time;

01.2. Оптимізація планування паралельного виконання ОП

3. Рівномірність розподілу обчислювальних ресурсів між процесами (fairness); (→ max)

4. Час очікування → максимальний сумарний або середній час, який процес проводить в стані очікування доступу до процесора (waiting time); (→ min)

01.3. Оптимізація планування паралельного виконання ОП

5. Відношення часу, витраченого на обчислення, до часу, витраченого на роботу диспетчера («ККД» диспетчера); визначається для коротко-термінового планування (short-term scheduler);

Приклад: через кожні 100 млс (=квант) на прийняття рішення витрачається 10 млс → «ККД» = (100/(100+10))*100 = 91%.

Конфлікт: пропускна здатність vs. затримка → пошук компромісу → евристичні алгоритми

01.4. Оптимізація планування паралельного виконання ОП

Методики вирішення оптимізаційних задач, пов’язаних з плануванням:

- Теорія черг = Теорія масового обслуговування (Queueing theory);
- Теорія розкладів (розділ дискретної математики) (Job shop scheduling);
- Динамічне програмування (Dynamic programming);
- Задачі на розподіл ресурсів (resource allocation).

01.5. Оптимізація планування паралельного виконання ОП

1) процеси → обчислювальне навантаження
2) процесори → обчислювальні ресурси
+
Задачі:
1) Load balancing: розподіл обчислювального навантаження між процесорами (ядрами процесора, машинами) + розподіл квантів часу окремого процесора; → Task allocation
2) Resource allocation→ на обчислення якої задачі виділити більше процесорів (ядер процесора, машин).

01.6. Оптимізація планування паралельного виконання ОП

1. Ручна диспетчеризація (manual scheduling).

2. Розробник задає жорсткий алгоритм послідовності доступу процесів до CPU.

3. Використовується в спеціальних системах, в яких відомо наперед який набір процесів буде виконуватись, на основі чого можна спробувати вирішити відповідну оптимізаційну задачу з повною інформацією.

02.1. Прив’язка ОП до процесора (Processor affinity)

- Processor affinity (CPU pinning) → «прив’язка» процеса до певного процесора в мульти-процесорних системах (зокрема в SMP-системах (Symmetric multiprocessing))

- ОС або користувач (програміст) визначає який процесор буде «рідним» (affinity) для даного процеса (тобто процес буде виконуватись лише на цьому процесорі або підмножині процесорів)

- Основна мета: зменшити кількість «промахів» при звертанні до кеша (cache misses) → після чергового переключення контексту процес повертається на «свій» процесор, в кеші якого лишаються його дані

02.2. Прив’язка ОП до процесора (Processor affinity)

Проблема: якщо два готових до виконання процеса «прив’язані» до одного процесора, а другий процесор вільний, то планувальник має прийняти рішення:

1) перенести один з процесів на вільний процесор (отримати виграш від Load balancing);

або

2) виконати процеси послідовно на першому процесорі (отримати виграш від Processor affinity)

02.3. Прив’язка ОП до процесора (Processor affinity): реалізація

- Варіант 1: для кожного процеса в черзі готовності (ready queue) визначається змінна-індикатор: 0 = не «прив’язаний», >0 = номер «рідного» процесора
- Варіант 2: в сучасних ОС використовується змінна affinity mask, яка задає підмножину «рідних» процесорів у вигляді бітової маски
- В момент вибору який процес на якому процесорі запустити планувальник віддає більшу перевагу відповідним «прив’язаним» процесам
- ОС Linux: «прив’язку» процеса до процесора можна встановити системним викликом:

```
sched_setaffinity(pid_t pid, size_t
cpusetsize, cpu_set_t *mask);
```

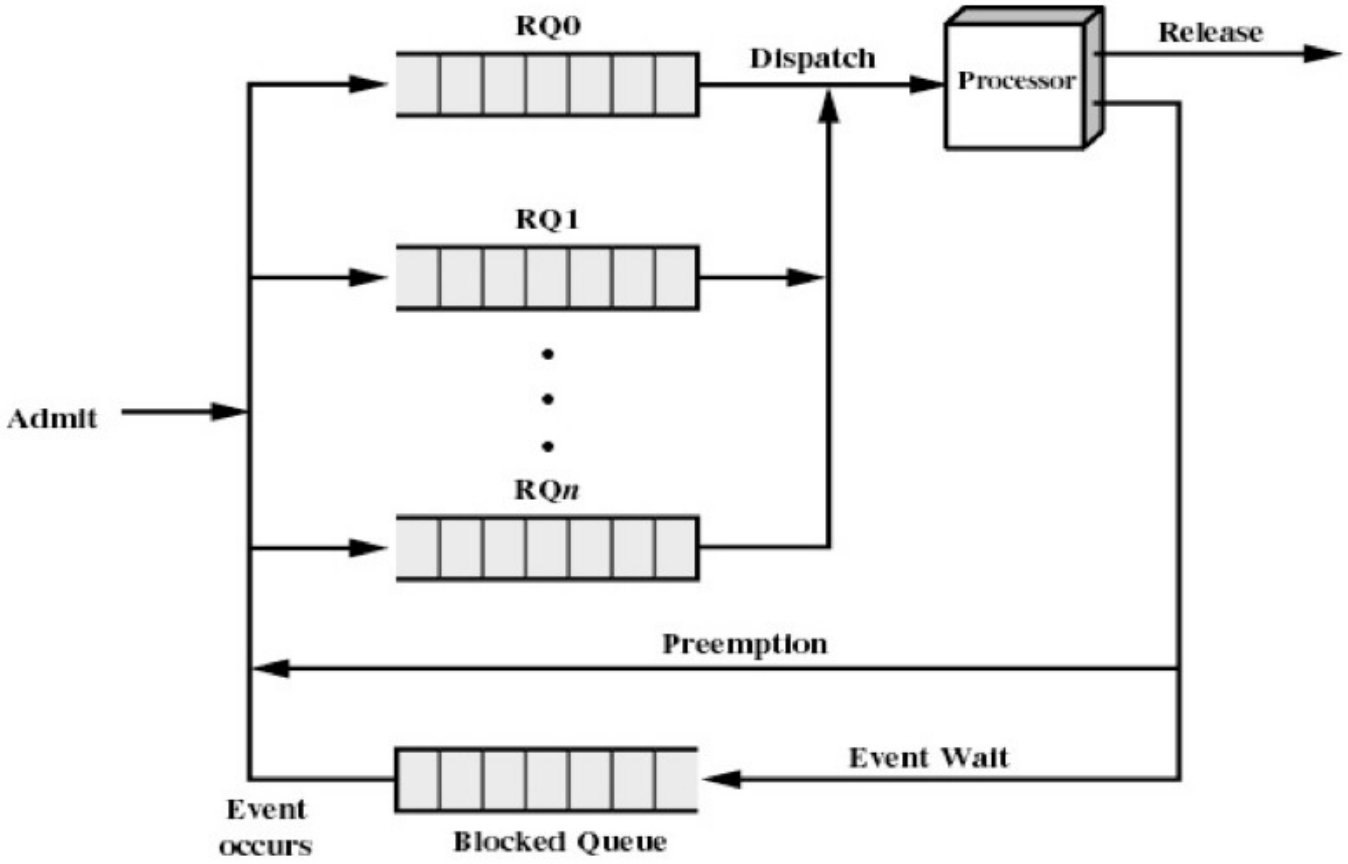
03.1. Планування паралельного виконання ОП в ОС UNIX

• Призначення: інтерактивні обчислювальні системи з розділенням в часі (time-sharing system + interactive environment)

• Цілі планування:
1) забезпечити прийнятний час відгуку на запит користувача (response time);
2) забезпечити «справедливість» розподілу часу CPU (fairness) між процесами.

• Алгоритм: багаторівнева черга зі зворотнім зв’язком (Multilevel Feedback Queue) + Round Robin в кожній черзі (кожній черзі відповідає своя група пріоритетів).

03.2. Планування паралельного виконання ОП в ОС UNIX



Узагальнена схема організації планування паралельного виконання ОП в ОС UNIX

03.3. Планування паралельного виконання ОП в ОС UNIX

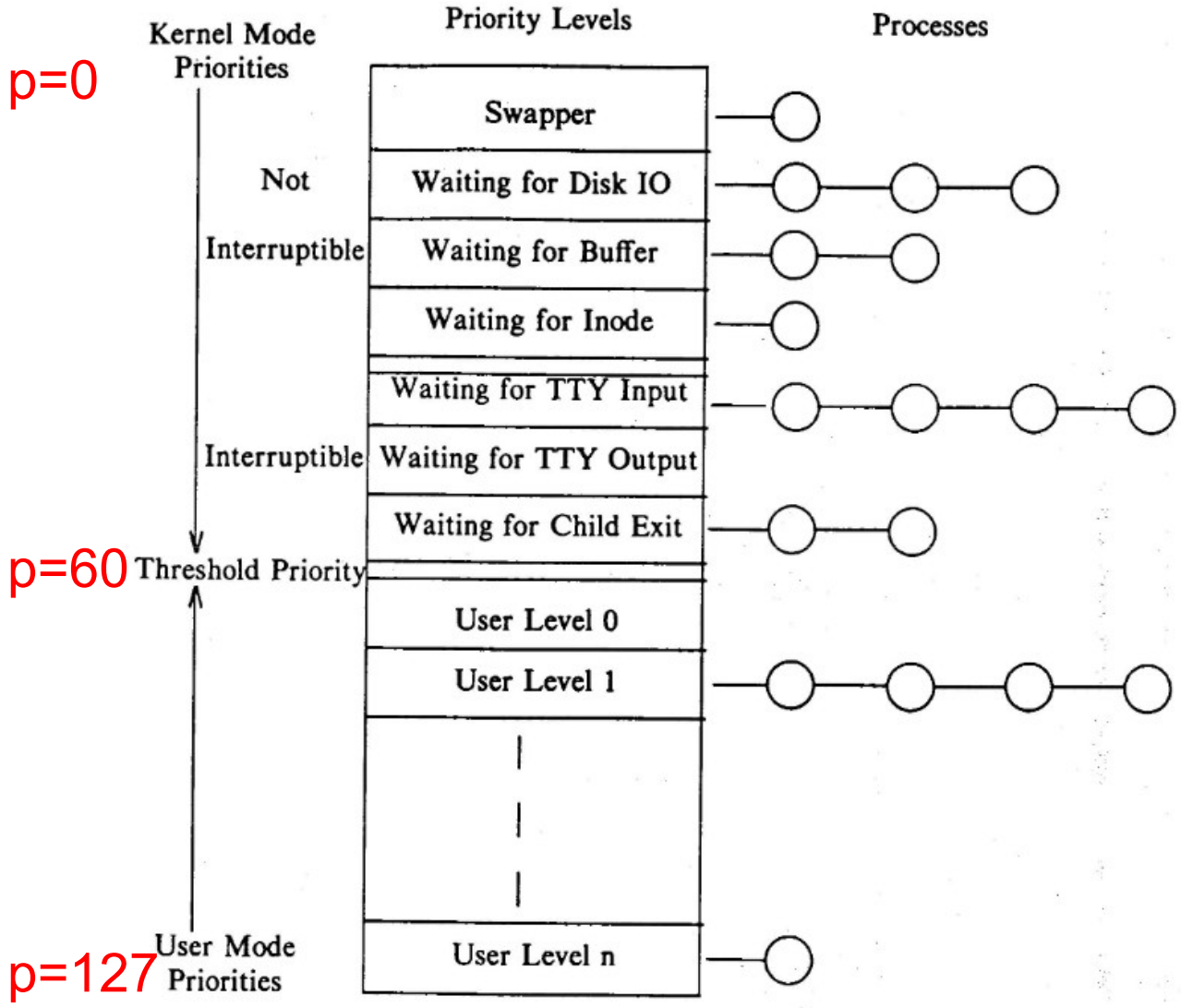
- Якщо поточний процес не блокується (призупиняється «сам») або не завершується на протязі 1 секунди, то він витісняється з CPU.
- Пріоритет визначається на основі типу процеса та «історії» його виконання (чим менше значення, тим більше пріоритет).
- Діапазон значень пріоритету: 0 — 127, пороговий пріоритет (base, P_USER): 60 (інші варіанти: 50, 65)
- Системні процеси (kernel mode): 0 — 59
- Користувацькі процеси (user mode): 60 — 127

03.4. Планування паралельного виконання ОП в ОС UNIX

Пріоритети в ОС UNIX розбиті на групи = рівні пріоритету:

– Операції своппінгу (Swapper) (найвищий пріоритет)
– Управління блочним вводом/виводом (Block I/O device control)
– Операції з файлами (File manipulation)
– Управління символьним вводом/виводом (Character I/O device control)
. . . . .
– Процеси користувача (User processes) (найнижчий пріоритет)

03.5. Планування паралельного виконання ОП в ОС UNIX

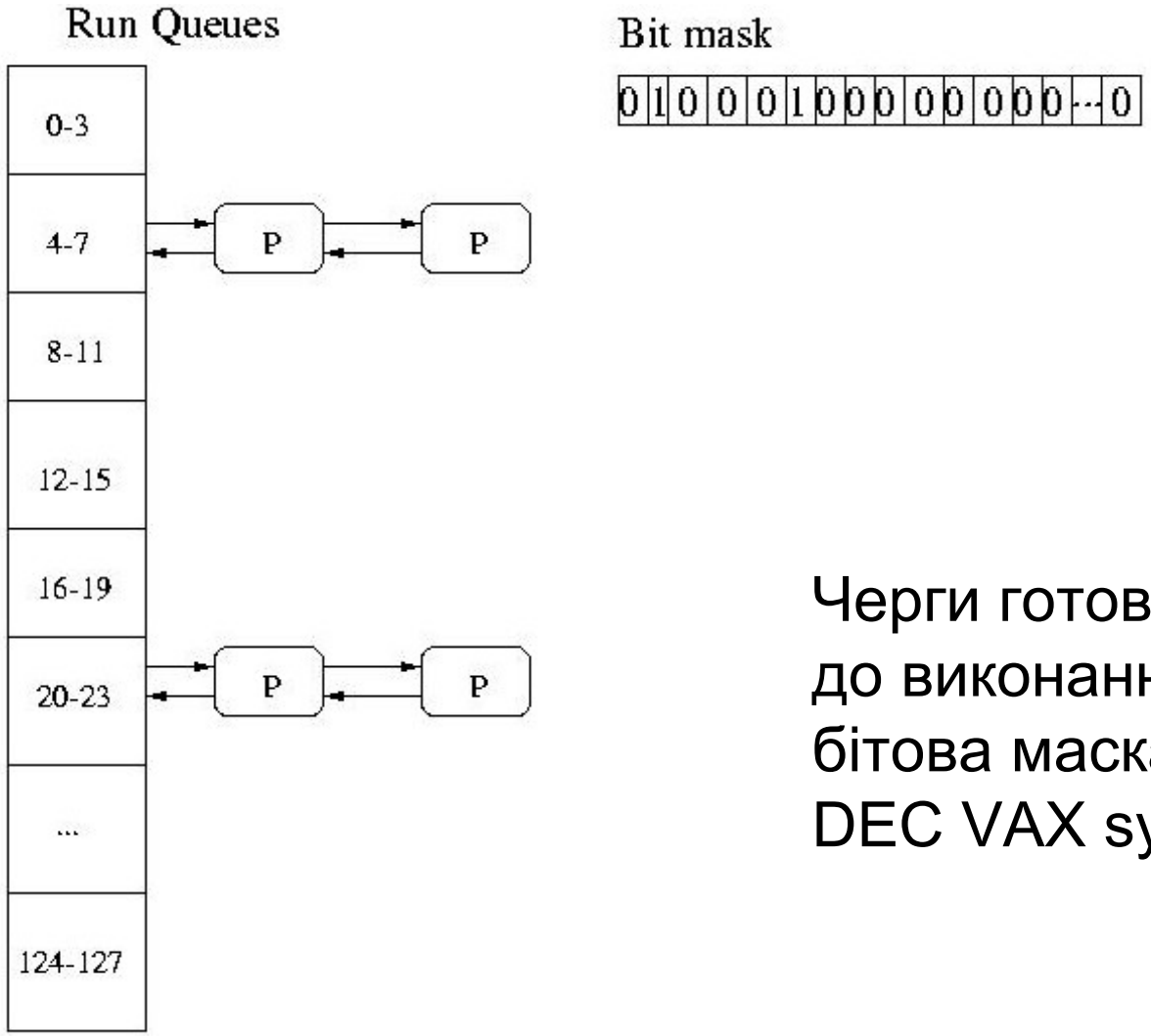


03.6. Планування паралельного виконання ОП в ОС UNIX

Приклад (DEC VAX system):

- Кількість рівнів=груп пріоритетів: 32
- У кожного рівня своя черга виконання (runqueue).
- Наявність процесів у чергах визначається бітовою маскою.
- На виконання у CPU обирається перший процес у черзі (Round Robin) з найвищим пріоритетом.
- При зміні пріоритету процес переміщається у відповідну чергу (Multilevel Feedback Queue).

03.7. Планування паралельного виконання ОП в ОС UNIX



Черги готовності до виконання та бітова маска в DEC VAX system

03.8. Планування паралельного виконання ОП в ОС UNIX

Для кожного процеса визначаються значення:

p_pri — поточний (актуальний) пріоритет
p_usrpri — пріоритет процесу в режимі користувача
p_cpu — оцінка використання процесом часу процесора (CPU usage) = оцінка CPU burst
p_nice — призначений користувачем показник важливості процеса (nice value)

Коли процес виконується в режимі користувача:

p_pri = p_usrpri

03.9. Планування паралельного виконання ОП в ОС UNIX

Пріоритет процеса змінюється в 4-ох випадках:

1) При переході процеса з режиму користувача в режим ядра (обробка системного виклика), система призначає йому відповідний пріоритет в діапазоні системних пріоритетів (p_pri = 0...59).

2) Процес в режимі ядра переходить в стан «Очікування»: система призначає йому пріоритет сна (sleep priority), який обирається з діапазону системних пріоритетів (p_pri = 0...59) і залежить лише від причини переходу процеса в даний стан. В такий спосіб надається перевага I/O-bound процесам перед CPU-bound процесами для забезпечення прийнятного часу відгуку. Принцип: чим більш низькорівнева операція, тим вищий пріоритет призначається.

03.10. Планування паралельного виконання ОП в ОС UNIX

3) Після повернення процеса з режиму ядра в режим користувача йому повертається відповідний «користувацький» пріоритет: p_pri = p_usrpri.

4) Для всіх процесів в режимі користувача через 1 секунду перераховується пріоритет (p_usrpri). Принцип: чим більше процесорного часу використовує процес (CPU-bound), тим менше його пріоритет (в такий спосіб забезпечюється «справедливість» (fairness) розподілу процесорного часу між процесами).

03.10. Планування паралельного виконання ОП в ОС UNIX

Перерахунок пріоритету p_usrpri:

1. Для процеса, який виконується у CPU, з кожним тіком (clock tick) (варіанти: 1/10 сек., 1/60 сек.) додається одиниця до значення p_cpu:

$$p\_cpu = p\_cpu + 1$$

Це робить обробник відповідного переривання від таймеру.

Результат: процес витіснений з CPU буде мати відносно велике значення p_cpu в порівнянні з іншими процесами. Значення p_cpu витісненого з CPU процесу - це свого роду оцінка його CPU Burst.

03.11. Планування паралельного виконання ОП в ОС UNIX

2. Decay («згасання», «розпад»): кожну 1 секунду для всіх процесів в режимі користувача в runqueue зменшується значення p_cpu. Базовий варіант («напіврозпад»):

$$p\_cpu = p\_cpu / 2$$

Мета:
1) обмежити «зверху» значення p_cpu;
2) забезпечити його поступове зменшення для процесів з великим CPU-burst в черзі готовності (runqueue) - механзім “старіння” (aging) для запобігання ситуації, коли процес з низьким пріоритетом ніколи не отримує доступу до процесора.

03.12. Планування паралельного виконання ОП в ОС UNIX

3. Розрахунок значення p_usrpri: кожну 1 секунду для всіх процесів в режимі користувача перераховується значення:

$$p\_usrpri = base + (p\_cpu/2) + p\_nice$$

base – пороговий пріоритет (P_USER, типове значення: 60) = найвищий пріоритет, який може мати процес в режимі користувача (коли p_cpu =0 та p_nice = 0);
(p_cpu/2) – оцінка системи, наскільки даний процес має бути «покараний» за «надмірне» споживання CPU (враховується історія виконання процесу);
p_nice – показник наскільки даний процес менш (більш) важливий ніж інші процеси (встановлюється користувачем, по замовченню = 0).

03.13. Планування паралельного виконання ОП в ОС UNIX

- Варіант розрахунку decay, який враховує поточну завантаженість системи:

$$p\_cpu = p\_cpu \cdot (2 \cdot load\_average)/(2 \cdot load\_average + 1)$$

load_average — середня кількість процесів в черзі готовності (runqueue) за останню секунду

- Варіант розрахунку пріоритету:

$$p\_usrpri = base + (p\_cpu/4) + (2 \cdot p\_nice)$$

04.1. Планування паралельного виконання ОП в ОС Linux

v.1.2: аналог UNIX-планувальника → multilevel queue (пріоритети) + Round-Robin

v.2.2: scheduling classes:
1) real-time tasks (пріоритети: 0...99 → незмінний пріоритет → nonpreemtible tasks (алгоритм FCFS));
2) non-real time tasks (пріоритети: 100...139 → статичний/динамічний пріоритет → nice value: [-20;+19]).
+ підтримка Symmetric multiprocessing (SMP)

Значення time quantum:
пріоритет 0 = 200 млс,
пріоритет 139 = 10 млс

04.2. Планування паралельного виконання ОП в ОС Linux

v.2.4: O(N) scheduler:

- проходить по циклу всі N задач в черзі готовності (run queue);

- для кожної задачі перераховує дві величини:
1) динамічний пріоритет (Goodness);
2) квант часу (time slice);

- fairness → aging → всім задачам в черзі готовності додається 1 до пріоритету (якщо сумма не перевищує MAX);

- недолік: чим більше задач, тим повільніше працює scheduler;

04.3. Планування паралельного виконання ОП в ОС Linux

- Цикл по відрізках часу (epoch) → кожна задача використовує свій квант
- Якщо задача не використала свій квант, то в наступному циклі її квант буде збільшено на 1/2 залишку
- Goodness computation (з використанням евристик): для всіх процесів у global runqueue (спільна черга для всіх процесорів у випадку SMP) в циклі розраховується показних «якості» (goodness) → процес з найбільшим значенням «якості» надходить у CPU

04.4. Планування паралельного виконання ОП в ОС Linux

Scheduling_strategy	Remaining quantum	Goodness
SCHED_FIFO, SCHED_RR	-	1000+priority
SCHED_OTHER	>0	Quantum+priority+1
SCHED_OTHER	=0	0

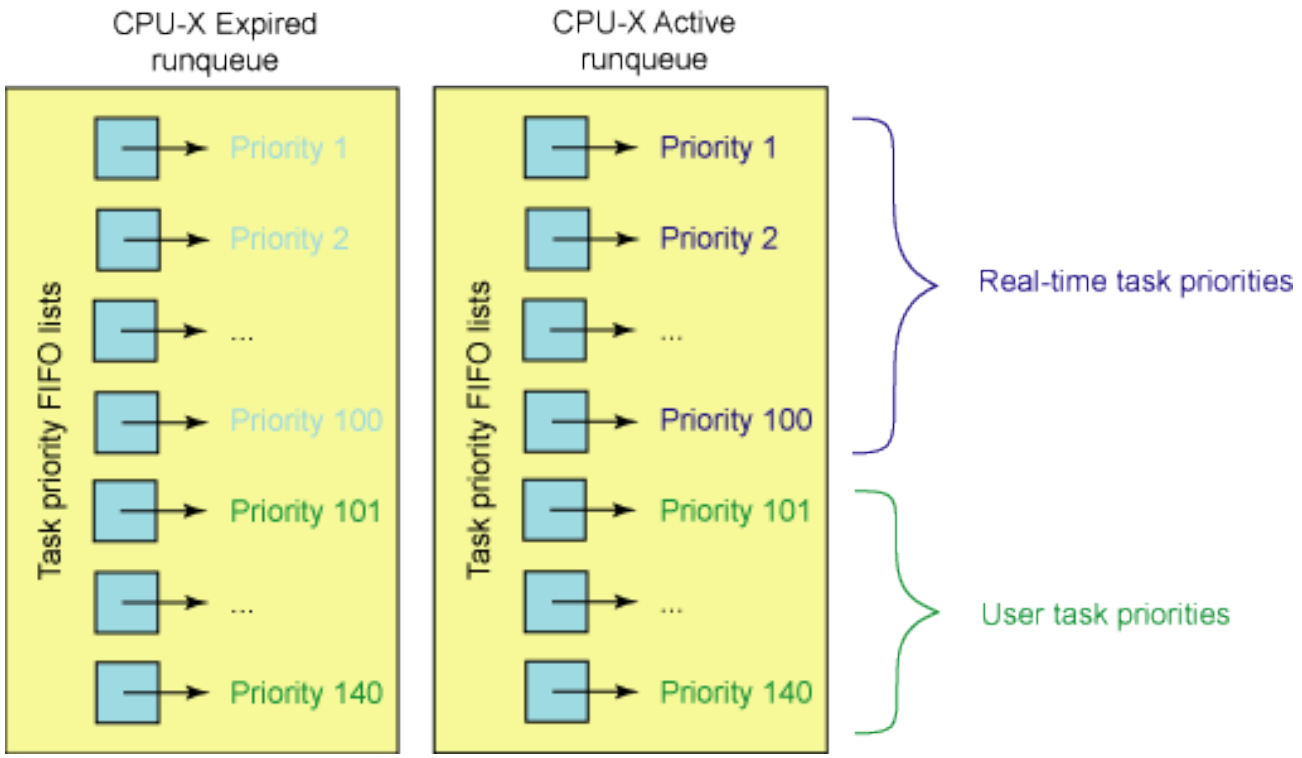
- I/O-bound процеси частіше недовикористовують свій квант (внаслідок зупинок на очікування операцій вводу/виводу), і відтак отримують більший пріоритет
- «+1» в таблиці → aging → fairness

04.5. Планування паралельного виконання ОП в ОС Linux

v.2.6: O(1) scheduler:

- Своя runqueue для кожного процесора (ядра)
- Незмінний (незалежний від N) час роботи алгоритму диспетчеризації → O(1)
- active runqueue → expired runqueue: коли процес вичерпав свій квант, він переходить у розряд expired
- Коли в active runqueue не лишається жодного процесу, відбувається переключення: expired runqueue стає active runqueue і навпаки

04.6. Планування паралельного виконання ОП в ОС Linux



O(1) scheduler: структура runqueue

04.7. Планування паралельного виконання ОП в ОС Linux

- пріоритет процесу перераховується один раз за одне спрацювання функції schedule() при переході процеса з active runqueue у expired runqueue
- fairness: по мірі вибування процесів з високим пріоритетом з active runqueue, доступ до CPU отримують процеси з низьким пріоритетом!
- переключення черг замість «традиційного» aging забезпечує часову складність роботи диспетчера O(1)

04.8. Планування паралельного виконання ОП в ОС Linux

- значення кванту (time slice) розраховується в момент заходу процесу у CPU → принцип: чим вище пріоритет, тим більше квант
- dynamic priority: для «звичайних» процесів відбувається перерахунок пріоритету в межах [-5;+5] до статичного пріоритету
- dynamic priority: використовуються складні евристики, основний принцип: чим більш інтерактивний процес (=I/O-bound → чим більше часу він проводить в I/O queue), тим вище його динамічний пріоритет
- load balancing → SMP → 200 млс → порівнюються черги до процесорів (ядер): процеси перекидаються в менш завантажену чергу + processor affinity

04.8. Планування паралельного виконання ОП в ОС Linux

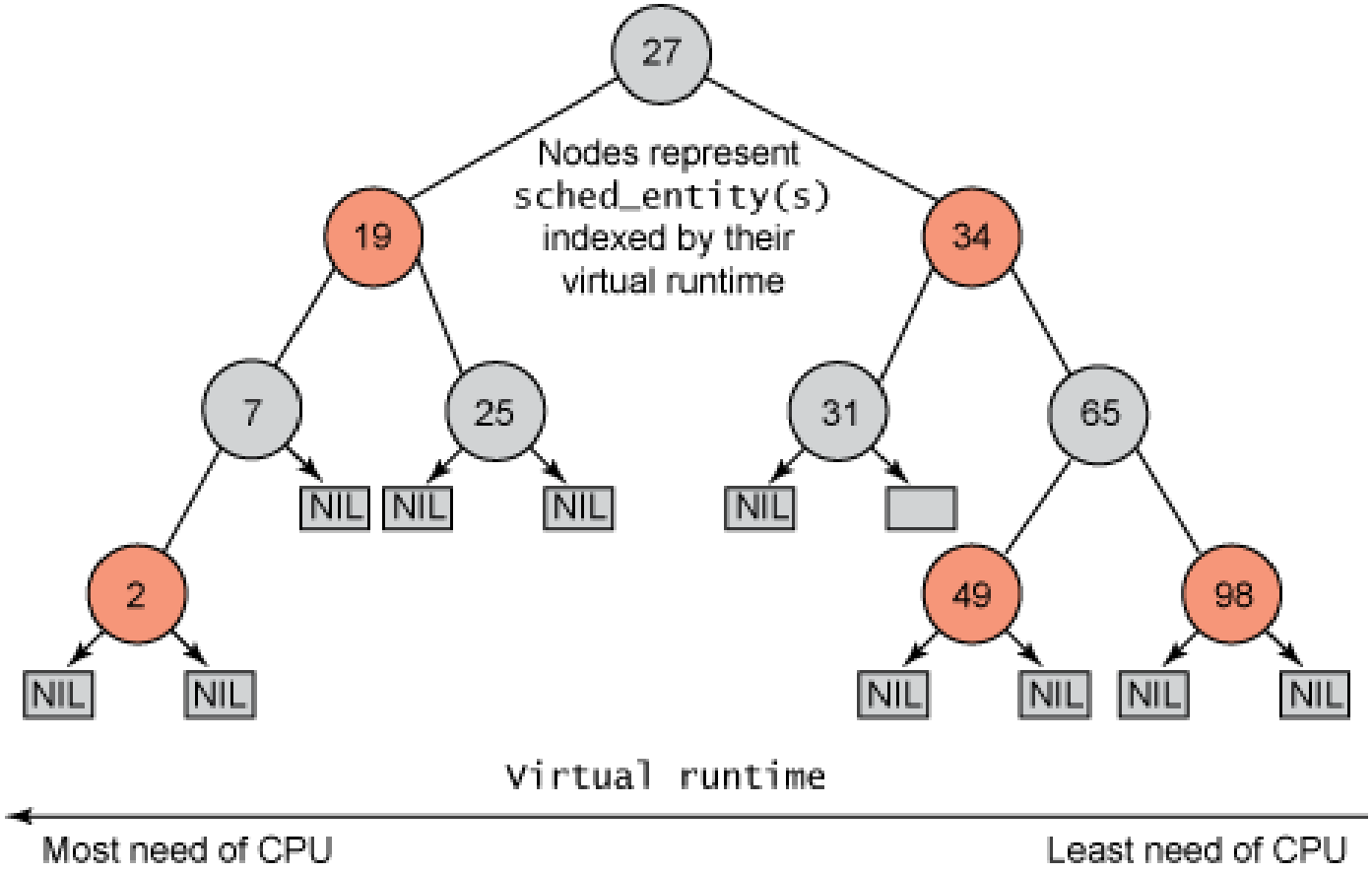
v.2.6.23: Completely Fair Scheduler (CFS) [Ingo Molnár]

- Оцінка складності (часу роботи): O(log N)
- Ідеальна «справедливість»: 2 процеса ділять ресурс процесора у співвідношенні 50%+50%, 4 процеса: 25%+25%+25%+25%
- CFS — це спроба реалізувати ідеальну «справедливість» (fairness) розподілу процесорного часу між процесами
- В CFS відсутні динамічні пріоритети, внаслідок чого відсутнє обчислювальне навантаження, яке породжував їх розрахунок.

04.9. Планування паралельного виконання ОП в ОС Linux

- Virtual run time (змінна vruntime) - відносний показник того, скільки процесорного часу потрібно задачі (оцінка CPU burst задачі).
- Замість runqueue у вигляді зв'язного списку використовується структура даних Red-Black tree (різновид self-balancing binary tree): жодна гілка в дереві ніколи не буде довшою за найкоротшу більше ніж в 2-а рази.
- Це забезпечує часову складність виконання операцій (пошук, додавання (insert) та видалення (delete)): O(log N). Разом з тим само-балансування дерева забезпечує високу швидкодію виконання операцій над даними.

04.10. Планування паралельного виконання ОП в ОС Linux



Приклад використання Red-Black tree в CFS

04.11. Планування паралельного виконання ОП в ОС Linux

- Місце задачі у RB-tree визначається величиною її vruntime, яке в термінології структури даних RB-tree називається ключем (key) відповідної вершини дерева.
- Для задачі, що повернулася у runqueue з CPU, виконується операція insert в RB-tree із збільшеним значенням key: key(t) = key(t-1) + CPU-burst * Δ,
  Δ - масштабучий коефіцієнт, який залежить від пріоритету задачі.
- Для задачі, що повернулася у runqueue з waitqueue (I/O queue), також виконується операція insert в RB-tree, як правило, з мінімальним значенням key (тобто така задача відразу потрапляє у ліву нижню вершину).

04.12. Планування паралельного виконання ОП в ОС Linux

- Нижня ліва вершина визначає задачу, яка «заходить» у CPU (і вибуває з дерева), відповідно в цей момент для неї виконується операція delete з RB-tree.
- Принцип:
  1) в середньому у CPU-bound задач великий CPU-burst, тоді як у I/O-bound задач CPU-burst малий;
  2) відтак CPU-bound задача повертається у RB-tree з більшим key, ніж I/O-bound задача.
- Тобто CPU-bound задача займає місце далі від лівої нижньої вершини ніж I/O-bound задача, коли вони повертаються у RB-tree.

04.13. Планування паралельного виконання ОП в ОС Linux

- Як пріоритет задачі впливає на її місце у RB-tree?
- Чим нижче пріоритет задачі, тим більше значення Δ (зокрема Δ>1), і тим більшим стає значення key(t).
- Чим вище пріоритет задачі, тим менше значення Δ (зокрема Δ<1), і тим меншим стає значення key(t).
- Масштаб часу більший для задач з нижчим пріоритетом, і навпаки - мастшаб часу менший для задач з викоским пріоритетом.
- Виконання всіх операцій (insert, delete) не перевищує встановлений час (виконується “in constant time”) завдяки використанню структури даних Red-Black tree.

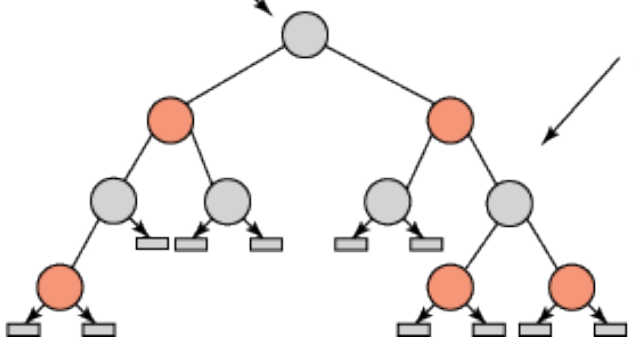
04.14. Планування паралельного виконання ОП в ОС Linux

```
struct task_struct {
  volatile long state;
  void *stack;
  unsigned int flags;
  int prio, static_prio normal_prio;
  const struct sched_class *sched_class;
  struct sched_entity se;
  ...
};
```

```
struct sched_entity {
  struct load_weight load;
  struct rb_node run_node;
  struct list_head group_node;
  ...
};
```

```
struct ofs_rq {
  ...
  struct rb_root tasks_timeline;
  ...
};
```

```
struct rb_node {
  unsigned long rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;
};
```



Структури даних, які використовує CFS для реалізації red-black tree

04.15. Планування паралельного виконання ОП в ОС Linux

1. Період диспетчеризації (scheduling period)

- p — період диспетчеризації (scheduling period, epoch, [schedule latency] period), значення якого зберігається в змінній sched_latency_ns. Величина p визначає «epoch duration (length) in nanoseconds (20 ms by default)», тобто відрізок часу, який CFS спробує розділити порівну між задачами, що готові до виконання.
- За логікою роботи CFS за цей час кожна задача хоча б один раз побуває у CPU. «This way no task gets starved for longer than a scheduling period.»
- Ця величина періодично перераховується в такий спосіб щоб відповідати біжучій кількості задач (nr_running, TASK_RUNNING).

04.16. Планування паралельного виконання ОП в ОС Linux

Принцип перерахунку p (змінна sched_latency_ns):

- Параметр sched_min_granularity_ns визначає в нс мінімальний часовий крок (по замовчуванню 4 млс), з яких складається величина sched_latency_ns. Параметр nr_running рівний кількості задач в runqueue.
- Якщо

  nr_running > sched_latency_ns / sched_min_granularity_ns

- тобто задач в системі більше ніж мінімальних за розміром квантів часу, які можна їм роздати на протязі періоду диспетчеризації, тоді необхідно збільшити величину періоду диспетчеризації p.

04.17. Планування паралельного виконання ОП в ОС Linux

Це робиться за формулою:

p = sched_min_granularity_ns * nr_running

Відтак значення періоду по замовченню (20 млс) буде збільшено якщо в runqueue буде більше п’яти задач (20 млс / 4 млс = 5).

```
$sudo sysctl -a | grep "sched" | grep -v "domain"
. . .
kernel.sched_latency_ns = 18000000              = 18 млс
kernel.sched_min_granularity_ns = 2250000       = 2.25 млс
. . .
```

«Note: Any given task's time slice is dependent on its priority and the number of tasks on the run queue.»

04.18. Планування паралельного виконання ОП в ОС Linux

2. Квант часу (time slice)

Розрахунок time_slice для задачі, яка обрана для заходу до CPU:

time_slice = p * (task_load / cfs_rq_load),

де

p — період диспетчеризації (повертається функцією sched_period()),

task_load — вага задачі в порівнянні з іншими, яка розраховується на основі пріоритету задачі (це значення зберігається у змінній se.load.weight),

cfs_rq_load — сума task_load всіх задач в runqueue (це значення зберігається у змінній cfs_rq.load.weigh).

04.19. Планування паралельного виконання ОП в ОС Linux

- В такий спосіб кожна задача отримує «справедливу» частину часу CPU (на протязі періоду диспетчеризації p),
- змасштабовану згідно величини її ваги (task_load) в порівнянні з іншими задачами (масштаб задається відношенням task_load / cfs_rq_load).
- Розрахунок task_load (значення зберігається в полі weight структури load_weight) базується на значенні пріоритету задачі (діапазон від 0 до 139) і виконується за допомогою масиву sched_prio_to_weight[].

04.20. Планування паралельного виконання ОП в ОС Linux

```
/*
 * Nice levels are multiplicative, with a gentle 10% change for every
 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
 * nice 1, it will get ~10% less CPU time than another CPU-bound task
 * that remained on nice 0.
 *
 * The "10% effect" is relative and cumulative: from _any_ nice level,
 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
 * If a task goes up by ~10% and another task goes down by ~10% then
 * the relative distance between them is ~25%.)
 */
const int sched_prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

04.21. Планування паралельного виконання ОП в ОС Linux

Приклад відображення пріоритету задачі у її task_load:

- A priority number of 120, which is the priority of a normal task, is mapped to a load of 1024, which is the value that the kernel uses to represent the capacity of a single standard CPU.
- The remaining values in the array are arranged such that the multiplier between two successive entries is ~1.25.
- This number is chosen such that if the priority number of a task is reduced by one level, its gets 10% higher share of CPU time than otherwise. Similarly if the priority number is increased by one level, the task will get a 10% lower share of the available CPU time.

04.22. Планування паралельного виконання ОП в ОС Linux

- If there are two tasks, A and B, running at a priority of 120(=0), the portion of available CPU time given to each task is calculated as:

  S(A) = S(B) = 1024/(1024*2) = 0.5

- However if the priority of task A is increased by one level to 121(=1), its load becomes:

  task_load(A) = (1024/1.25) = ~820

- (Recall that higher the number, lesser is the load). Then, task A's portion of the CPU becomes:

  S(A) = 820/(1024+820)) = ~0.45

- while task B will get:

  S(B) = (1024/(1024+820)) = ~0.55

- This is a 10% decrease in the CPU time share for Task A.

04.23. Планування паралельного виконання ОП в ОС Linux

3. Віртуальний час задачі (vruntime)

- «In practice, the virtual runtime of a task is its actual runtime normalized to the total number of running tasks.»

- Перерахунок vruntime для задачі, яка повертається з CPU у runqueue

  vruntime = vruntime + t * (NICE_0_LOAD / task_load)

- де
  t — час, який провела задача в CPU,
  NICE_0_LOAD – вага задачі з «нормальним» пріоритетом (priority =120(=0), nice=0), яка грає роль «опорної точки»,
  task_load – вага даної задачі.

04.24. Планування паралельного виконання ОП в ОС Linux

$$\Delta = NICE\_0\_LOAD / task\_load$$

- Більш важлива задача (з меншим значенням nice) буде мати більшу вагу task_load ніж задача з «нормальним» пріоритетом.
- За рахунок цього $\Delta < 1$, і віртуальний час (vruntime), який для неї визначається, буде меншим.
- Відповідно для менш важливої задачі з меншою вагою task_load ніж задача з «нормальним» пріоритетом все навпаки: $\Delta > 1$, і віртуальний час (vruntime), який для неї визначається, буде більшим.
- Тобто для більш важливих задач масштаб часу зменшується ($\Delta<1$), а для менш важливих задач масштаб часу збільшується ($\Delta>1$).

04.25. Планування паралельного виконання ОП в ОС Linux

Алгоритм роботи CFS

1. Вибрати задачу з лівого нижнього вузла RB-tree, тобто задачу з найменшим значенням vruntime. Посилання на цей вузол зберігається у змінній cfs_rq.rb_leftmost (його повертає функція __pick_first_entity()), тому часова складність цієї операції O(1).

2. Розрахувати time_slice для обраної задачі і запустити її на CPU на цей час.

3. Після закінчення time_slice, перерахувати значення vruntime задачі. Якщо воно менше min_vrtime, то перейти на п.2, інакше повернути задачу у RB-tree (операція insert) і перейти на п.1. Якщо задача покинула CPU до завершення time_slice, то перерахувати її vruntime і перейти до п.1.

04.26. Планування паралельного виконання ОП в ОС Linux

Принцип: нове значення vruntime перемістить задачу ближче до правої сторони дерева, що дасть можливість іншим задачам поступово зміститись вліво і врешті решт стати нижнім лівим вузлом дерева.

4. При потребі перерахувати значення sched_latency_ns для того, щоб воно відповідало поточній кількості задач в runqueue. При надходженні у runqueue новної задачі, розпочинається новий період диспетчеризації (scheduling period).

5. Значення vruntime задачі, яка повертається до runqueue з waitqueue, (операція insert в RB-tree) призначається як максимальне з двох значень: 1) останнього значення vruntime, яке задача мала коли покинула runqueue і перейшла у waitqueue; 2) поточне значення min_vruntime. Як правило, це min_vruntime, оскільки за час, який задача “спить” у waitqueue, значення vruntime усіх інших задач і відповідно поточне значення min_vruntime монотонно зростають.