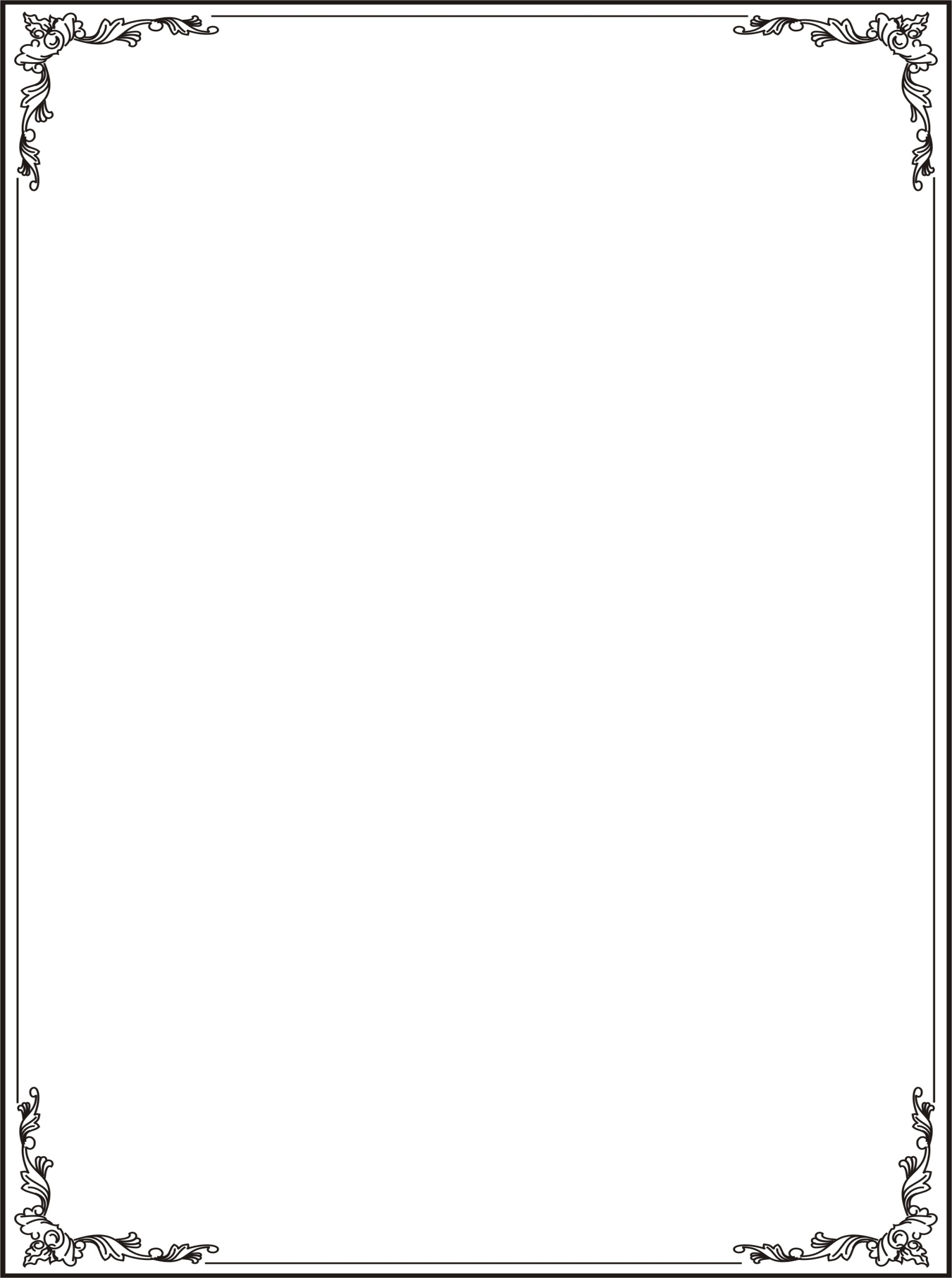
**ĐẠI HỌC QUỐC GIA TP.HCM**



**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH**

-----🙞🙜🕮🙞🙜-----



**BÁO CÁO BÀI TẬP LỚN**

**Môn học: Hệ điều hành**

**Giáo viên hướng dẫn:** Đoàn Việt Hưng

**Sinh viên thực hiện:**

|  |  |
| --- | --- |
| Tên | MSSV |
| Huỳnh Tấn Phúc | 51202787 |
| Nguyễn Thanh Tùng | 51204401 |

TP. HỒ CHÍ MINH, THÁNG 05 NĂM 2014

MỤC LỤC

[I) Tổng quan 3](#_Toc389326057)

[II) Data structure (Cấu trúc dữ liệu) 4](#_Toc389326058)

[1) Struct task\_struct 4](#_Toc389326059)

[2) Struct rq 5](#_Toc389326060)

[3) Struct Sched\_class 5](#_Toc389326061)

[III) Hàm schedule() 7](#_Toc389326062)

[IV) Bộ định thời Dummy 8](#_Toc389326063)

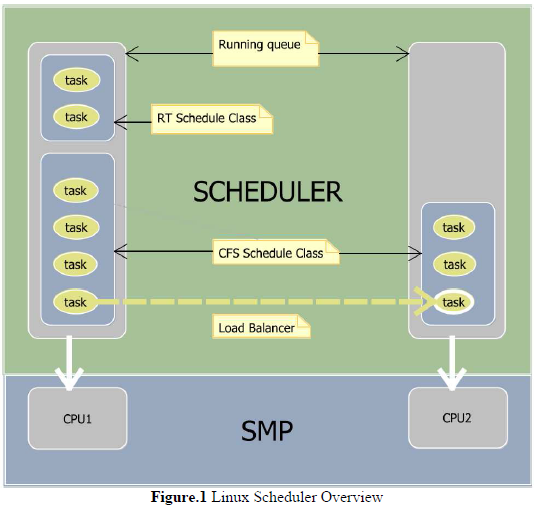
[1) Giới thiệu 8](#_Toc389326064)

[2) Chức năng từng hàm trong bộ định thời dummy 8](#_Toc389326065)

[V) Tài liệu tham khảo 13](#_Toc389326066)

[VI) Phụ lục 14](#_Toc389326067)

**I) Tổng quan**



Bộ định thời trong Linux gồm có:

**Running queue:**

* Mỗi CPU có một running queue (rq). Mỗi running queue chứa một danh sách các process (tiến trình) đang đợi để sử dụng CPU đó.

**Schedule class:**

* Linux scheduler là một module có khả năng kích hoạt nhiều giải thuật khác nhau cho những loại process khác nhau. Modul này được gọi là *scheduler classes.*
* Mỗi schedule class có một độ ưu tiên.

**Load balance:**

* Mỗi CPU có một running queue riêng của mình, các running queue này có thể không cân bằng, một bên có nhiều process, một bên có quá ít process. Load balance sẽ được gọi khi hệ thống yêu cầu định thời các process, nếu các running queue không được cân bằng, load balance sẽ cố gắng làm cho nó cân bằng để tăng hiệu suất sử dụng CPU.

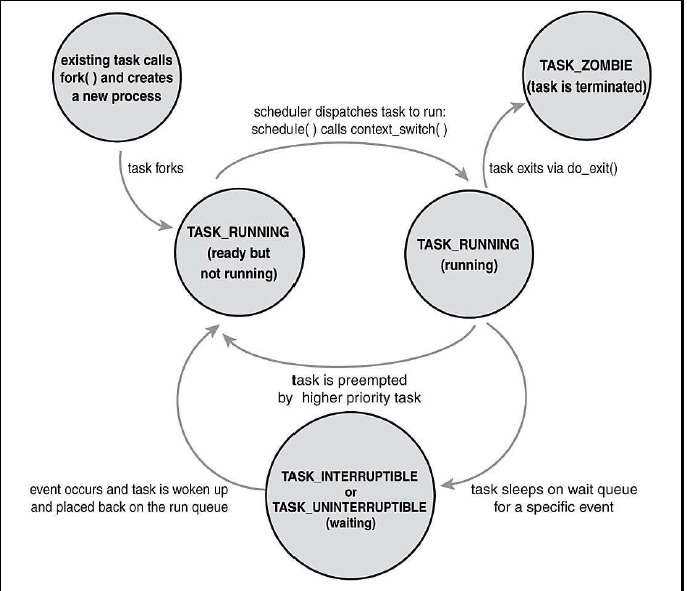
**II) Data structure (Cấu trúc dữ liệu)**

Hai trong số những cấu trúc dữ liệu quan trọng nhất trong bộ định thời là struct task\_struct và struct rq. Nhiều phương thức trong bộ định thời làm việc với hai cấu trúc này.

**1) Struct task\_struct**

* Mỗi một process trong hệ thống được diễn tả bằng một task\_struct. Do đó, kiểu dữ liệu này phải có khả năng nắm bắt được tất các thông tin của process.
* Khi một process hay một thread được tạo, kernel sẽ cấp một task\_struct cho nó.
* Một số field quan trọng trong task\_struct là:
* **State:** đây là field mô tả trạng thái hiện tại của process.

TASK\_RUNNING: là một process đang chạy hoặc đang đợi trên running queue.



TASK\_INTERRUPTIBLE: process đang ngủ (bị blocked), nó đợi điều kiện để được tồn tại.

TASK\_UNINTERRUPTIBLE: process đang ngủ, nó chờ nhận tín hiệu để được đánh thức.

TASK\_ZOMBIE: tiến trình hoàn tất, nhưng process cha vẫn chưa gọi thông qua wait() sysem call.

TASK STOPPED: sự thực thi của process hiện tại bị ngưng; tiến trình không phải đang chay và cũng không đủ điều kiện để chạy.

* **static\_prio:** đây là priority tĩnh của process. Giá trị của field này không thay đổi.
* **prio:** đây là priority động của process.
* **normal\_prio:** nó giữ priority dự kiến của process. Trong hầu hết trường hợp, đối với non real-time processes, giá trị của normal\_prio và static\_prio là như nhau. Đối với real-time processes, giá trị của normal\_prio có thể được tăng để tránh deadlock khi truy cập vào vùng tranh chấp.
* **rt\_priority:** đây là field được dung cho real-time process. Nó giữ giá trị real-time priority.
* **sched\_class:** nó là con trỏ trỏ đến schedule class.
* **sched\_entity:** là con trỏ đến CFS schedule entity.
* **sched\_rt\_entity:** là con trỏ đến RT schedule entity.
* **policy:** giữ giá trị của chính sách định thời.

**2) Struct rq**

* Tại thời điểm khởi động, Linux tạo một running queue có kiểu dữ liệu là struct rq cho mỗi CPU.
* Mỗi running queue giữ những thong tin sau:
* **nr\_running:** số của runnable task.
* **nr\_switches:** số lần chuyển đổi.
* **cfs:** cấu trúc running queue của bộ định thời CFS.
* **rt:** cấu trúc running queue của của bộ định thời RT.
* **next\_balance:** nhãn thời gian đến lần kiểm tra load balance tiếp theo.
* **curr:** con trỏ đến tiến trình đang chạy trên running queue.
* **idle:** con trỏ đến tiến trình “nhàn rỗi” trên running queue.
* **lock:** spin lock của running queue.

**3) Struct Sched\_class**

**Scheduling class**

Bộ định thời Linux là một mô đun, nó cho phép các thuật toán/ chính sách khác nhau có thể định thời các loại nhiệm vụ khác nhau. Mỗi thuật giải định thời gọi scheduling class. Scheduling class có một cái khung về giao diện để nó có thể xử lý các task theo sự tiến hành của thuật giải.

* Struct sched\_class có thể được tìm thấy trong include/linux/sched.h

struct sched\_class {

const struct sched\_class \*next;

void (\*enqueue\_task) (struct rq \*rq, struct task\_struct \*p, int flags);

void (\*dequeue\_task) (struct rq \*rq, struct task\_struct \*p, int flags);

void (\*yield\_task) (struct rq \*rq);

bool (\*yield\_to\_task) (struct rq \*rq, struct task\_struct \*p, bool preempt);

void (\*check\_preempt\_curr) (struct rq \*rq, struct task\_struct \*p, int flags);

struct task\_struct \* (\*pick\_next\_task) (struct rq \*rq);

void (\*put\_prev\_task) (struct rq \*rq, struct task\_struct \*p);

#ifdef CONFIG\_SMP

int (\*select\_task\_rq)(struct task\_struct \*p, int sd\_flag, int flags);

void (\*migrate\_task\_rq)(struct task\_struct \*p, int next\_cpu);

void (\*pre\_schedule) (struct rq \*this\_rq, struct task\_struct \*task);

void (\*post\_schedule) (struct rq \*this\_rq);

void (\*task\_waking) (struct task\_struct \*task);

void (\*task\_woken) (struct rq \*this\_rq, struct task\_struct \*task);

void (\*set\_cpus\_allowed)(struct task\_struct \*p,

const struct cpumask \*newmask);

void (\*rq\_online)(struct rq \*rq);

void (\*rq\_offline)(struct rq \*rq);

#endif

void (\*set\_curr\_task) (struct rq \*rq);

void (\*task\_tick) (struct rq \*rq, struct task\_struct \*p, int queued);

void (\*task\_fork) (struct task\_struct \*p);

void (\*switched\_from) (struct rq \*this\_rq, struct task\_struct \*task);

void (\*switched\_to) (struct rq \*this\_rq, struct task\_struct \*task);

void (\*prio\_changed) (struct rq \*this\_rq, struct task\_struct \*task,

int oldprio);

unsigned int (\*get\_rr\_interval) (struct rq \*rq,

struct task\_struct \*task);

#ifdef CONFIG\_FAIR\_GROUP\_SCHED

void (\*task\_move\_group) (struct task\_struct \*p, int on\_rq);

#endif

};

- Ngoại trừ hàm đầu tiên, tất cả các hàm còn lại đều là hàm con trỏ, được dùng bởi khung bộ định thời để gọi các chính sách tương ứng khi thực thi.

- Tất cả bộ định thời mà tồn tại trong kernel là một list, được sắp xếp bằng độ ưu tiên do class scheduling quyết định.

**stop\_sched\_class → rt\_sched\_class → fair\_sched\_class → idle\_sched\_class → NULL**

- Bộ định thời Dummy được chèn vào **trước** bộ định thời **idle**.

**Một số phương thức quan trọng của struct sched\_class:**

* **enqueue\_task:** thêm một process mới vào run queue. Điều này xuất hiện khi một process thay đổi trạng thái từ ngủ sang chạy.
* **dequeue\_task:** lấy một process ra khỏi run queue. Diều này xảy ra khi process này chuyển trạng thái từ chạy sang không chạy hoặc do kernel chỉ định lấy nó ra khỏi run queue vì nhiều lý do – ví dụ do priority của nó cần được thay đổi.
* **yield\_task:** Hàm này được gọi khi task tự nguyện trả lại CPU.
* **check\_preempt\_curr:** khi một task vào running queue, xét một vài điều kiện kiểm tra xem có nên preempt task hiện tại không.
* **pick\_next\_task:** hàm này chọn một task tiếp theo để chạy nếu nó hợp lệ.
* **put\_prev\_task:** hàm này được gọi trước khi task hiện thời đang thực thi bị thay thế bởi 1 task nào đó.
* **task\_tick:** dược gọi bởi bộ định thời mỗi khi nó được kích hoạt.
* **new\_task:** cho phép cài đặt kết nối giữa fork system call and scheduler. Mỗi lần một task mới được tạo, bộ định thời sẽ thông báo diều đó với new\_task().

**III) Hàm schedule()**

Đây là phương thức quan trọng nhất của bộ định thời. Nó chịu trách nhiệm chọn task tiếp theo và chuyển đổi task hiện tại với task kế tiếp. Phương thức này được thực hiện khi:

* Tiến trình tình nguyện trả lại CPU.
* Tiến trình đợi tín hiệu xảy ra hoặc muốn sleep.
* Timer task xảy ra thông qua scheduler\_tick().
* Và các trường hợp khác.

Các bước tiến hành:

* **Bước 1:** disables preemption. Trong suốt thời gian thực thi bộ định thời, một tiến trình bị preempt thì phải bị disabled.
* **Bước 2:** lấy thông tin từ running queue dựa trên trạng thái hiện tại của proccesor. Chúng ta gọi smp\_processor\_id() để lấy CPU hiện tại. Gọi cpu\_runqueue để lấy rq của CPU này. Sau đó gọi release\_kernel\_lock() để thoát khỏi kernel lock từ task hiện tại và nhận được khóa của rq hiện tại.
* **Bước 3:** thực thi phương thức pre\_schedule(). Gọi lại phương thức pre\_schedule() đã được định nghĩa trong struct sched\_class. Đây là cái hook cho việc thực thi bất cứ bất cứ chính sách định thời logic nào trước khi bộ định thời gọi pick\_next\_task(). Hiện nay, chỉ RT schedule class thực thi phương thức này.
* **Bước 4:** thực thi pick\_next\_task().
* **Bước 5:** Kiểm tra xem task hiện tại có giống task kế tiếp. Nếu giống thì không có chuyển ngữ cảnh thực thi post\_schedule(). Nếu không giống nhau thì chuyển ngữ cảnh được yêu cầu sau đó cũng gọi post\_schedule().
* **Bước 6:** post\_schedule() là cái móc, nó cho phép bộ định thời RT đặt real-time task lên processor hiện tại.
* **Bước 7:** tạm thu lại khóa trên task hiện tại, đó có thể là task ban đầu hoặc task tiếp theo và cho phép preempt. Kiểm tra lại code nếu chúng ta cần định thời lại. Nếu như vậy chúng ta cần trở lại nhãn tương ứng và làm lại thời biểu.

**IV) Bộ định thời Dummy**

*Chúng em đã thay đổi lại tất cả các hàm trong bộ định thời dummy so với lần báo cáo đầu tiên.*

**1) Giới thiệu**

- Bộ định thời dummy sẽ tiến hành định thời các process có kernel priority nằm trong khoảng 131 -> 135, tức có nice priority trong khoảng 11->15.

- Cơ chế của bộ định thời dummy là sẽ tiến hành preempt process đang chạy nếu có một process có độ ưu tiên cao hơn đến hoặc process đang chạy hết timeslice. Ta cũng cần có một phương pháp để chống việc trì hoãn vô hạn định (starvation) của các process có độ ưu tiên thấp bằng cách tăng độ ưu tiên nếu các process này đã chờ được một khoảng thời gian là 3 lần timeslice.

**2) Chức năng từng hàm trong bộ định thời dummy**

* **enqueue\_task\_dummy**

- Khi một process thay đổi trạng thái từ ngủ sang chạy, hàm *enqueue\_task\_dummy* sẽ thêm process mới này vào cuối dummy run queue, tăng biến *nr\_running* lên 1 và khởi tạo các giá trị *timeslice* và *age\_threshold* là 0.

Code:

/\*

\* Add the task to the tail of dummy run queue and initialize the timeslice and age\_threshold is zero

\*/

static void enqueue\_task\_dummy(struct rq \*rq, struct task\_struct \*p, int flags)

{

\_enqueue\_task\_dummy(rq, p);

inc\_nr\_running(rq);

}

* **dequeue\_task\_dummy**

- Hàm *dequeue\_task\_dummy* sẽ lấy một process ra khỏi dummy run queue.

Code:

/\*

\* Delete a task form the dummy run queue

\*/

static void dequeue\_task\_dummy(struct rq \*rq, struct task\_struct \*p, int flags)

{

\_dequeue\_task\_dummy(p);

dec\_nr\_running(rq);

}

* **yield\_task\_dummy**

- Khi một process tự nguyện trả lại CPU, hàm *yield\_task\_dummy* sẽ được gọi.

- Chức năng của hàm là gán các giá trị *timeslice* và *age\_threshold* của process muốn trả lại CPU trở về 0, sau đó đặt nó ở cuối dummy run queue để chắc chắn rằng process này sẽ không được thực thi ở lần kế tiếp.

Code:

/\*

\* When a process wants to relinquish control of the processor voluntarily, it can use the

\* sched\_yield system call. This triggers yield\_task to be called in the kernel.

\*/

static void yield\_task\_dummy(struct rq \*rq)

{

/\*

\* Reset the timeslice and age\_threshold of the current task is zero

\*/

rq->curr->dummy\_se.timeslice = 0;

rq->curr->dummy\_se.age\_threshold = 0;

/\* Delete and Add the current task to the tail of dummy run queue to ensure that

\* the current task will be end of the dummy run queue

\*/

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

}

* **check\_preempt\_curr\_dummy**

- Khi một process được thêm vào dummy run queue, hàm *check\_preempt\_curr\_dummy* sẽ được gọi để kiểm tra xem process mới có thể preempt process đang chạy hay không, nếu có, ta sẽ tiến hành thay thế process đang chạy bằng process mới.

Code:

/\*

\* check\_preempt\_curr is used to preempt the current task with a newly woken task if this is

\* necessary.

\*/

static void check\_preempt\_curr\_dummy(struct rq \*rq, struct task\_struct \*p, int flags)

{

/\*

\* If the task has priority higher than the priority of current task, the currently executing task will be preempted

\* Else, we add the task p to the tail of dummy run queue

\*/

if(p->prio < rq->curr->prio)

{

set\_tsk\_need\_resched(rq->curr);

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

}

else

{

enqueue\_task\_dummy(rq, p, 0);

}

}

* **pick\_next\_task\_dummy**

- Chức năng của hàm là chọn một process có độ ưu tiên cao nhất trong dummy run queue để chạy, nếu có nhiều process có cùng độ ưu tiên thì ta sẽ chọn process đã chờ lâu nhất.

Code:

/\*

\* Selects the next task that is supposed to run

\* Pick up the highest-prio task:

\*/

static struct task\_struct \*pick\_next\_task\_dummy(struct rq \*rq)

{

/\*

\* Choose the task has highest priority

\*/

struct dummy\_rq \*dummy\_rq = &rq->dummy;

struct list\_head \*temp\_list\_head = NULL;

struct sched\_dummy\_entity \*temp\_entity = NULL;

struct task\_struct \*temp\_task = NULL;

struct task\_struct \*task\_has\_max\_prio = NULL;

int max\_prio;

/\* If dummy run queue not empty, we find the task has highest priority,

\* Else, we return NULL

\*/

if (!list\_empty(&dummy\_rq->queue))

{

temp\_list\_head = dummy\_rq->queue->next;// temp\_list\_head point to the first

entry

temp\_entity = list\_entry(temp\_list\_head, struct sched\_dummy\_entity, run\_list);// get the entity that contains run\_list = temp\_list\_head

temp\_task = dummy\_task\_of(temp\_entity);// get the task that contains

dummy\_se = temp\_entity

task\_has\_max\_prio = temp\_task;

max\_prio = task\_has\_max\_prio->prio;

temp\_list\_head = temp\_list\_head->next;

while(temp\_list\_head != &dummy\_rq->queue)// loop throught all the task and

find the task has highest priority

{

temp\_entity = list\_entry(temp\_list\_head, struct sched\_dummy\_entity, run\_list);

temp\_task = dummy\_task\_of(temp\_entity);

if(temp\_task->prio < max\_prio)

{

task\_has\_max\_prio = temp\_task;

max\_prio = task\_has\_mnax\_prio->prio;

}

temp\_list\_head = temp\_list\_head->next;

}

return task\_has\_max\_prio;

}

else

{

return NULL;

}

}

* **put\_prev\_task\_dummy**

- Trước khi process hiện thời được thay thế bởi một process khác, chúng ta sẽ di chuyển process đang chạy vào cuối dummy run queue để chắc chắn rằng process đang chạy sẽ không được thực thi ở lần kế tiếp.

Code:

/\*

\* put\_prev\_task is called before

\* the currently executing task is replaced with another one.

\*

\* We delete it from dummy queue and then add it to tail of dummy queue

\*/

static void put\_prev\_task\_dummy(struct rq \*rq, struct task\_struct \*prev)

{

struct sched\_dummy\_entity \*dummy\_se = &rq->curr->dummy\_se; // take entity of the currently executing task

struct list\_head \*queue = &rq->dummy.queue; // take dummy queue

list\_move\_tail(&dummy\_se->run\_list, queue); // see list.h for more information line 164

}

* **set\_curr\_task\_dummy**

- Khi chính sách định thời của process đang thực thi bị thay đổi, hàm set\_curr\_task\_dummy sẽ delete process ra khỏi dummy run queue.

Code:

/\*

\* set\_curr\_task is called when the scheduling policy of the current task is changed.

\*/

static void set\_curr\_task\_dummy(struct rq \*rq)

{

dequeue\_task\_dummy(rq, rq->curr, 0);

}

* **task\_tick\_dummy**

- Mỗi khi hàm *task\_tick\_dummy* được gọi, timeslice của process đang chạy sẽ được cập nhật. Nếu process đang chạy sử dụng hết timeslice thì sẽ bị preempt và được thay thế bằng một process khác.

- Bên cạnh đó, hàm task\_tick cũng liên tục cập nhật thời gian chờ của các process đang ở trong dummy run queue, nếu thời gian chờ của các process quá lâu, chúng ta sẽ tăng độ ưu tiên của các process lên một bậc.

Code:

/\*

\* task\_tick\_dummy is called by the periodic scheduler each time it is activated.

\*/

static void task\_tick\_dummy(struct rq \*rq, struct task\_struct \*curr, int queued)

{

// Update the current timeslice

curr->dummy\_se->timeslice++;

// If the current task exhaust the timeslice, the currently executing task can be replaced with another one.

if(curr->dummy\_se->timeslice == DUMMY\_TIMESLICE)

{

curr->dummy\_se->timeslice = 0;

set\_tsk\_need\_resched(curr);

// We dequeue and enqueue the current task to ensure that the current task is moved to the tail of queue

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

return;

}

// Update the age\_threshold of all the waiting task

struct dummy\_rq \*dummy\_rq = &rq->dummy;

struct list\_head \*temp\_list\_head = dummy\_rq->queue.next;

struct sched\_dummy\_entity \*temp\_entity = NULL;

struct task\_struct \*temp\_task;

if (!list\_empty(&dummy\_rq->queue))

{

// Loop through all the task and update the age\_threshold

// If the age\_threshold equal DUMMY\_AGE\_THRESHOLD, we increase the priority by 1

while(temp\_list\_head != &dummy\_rq->queue)

{

temp\_entity = list\_entry(temp\_list\_head, struct sched\_dummy\_entity, run\_list);

temp\_entity->age\_threshold++;

if(temp\_entity->age\_threshold == DUMMY\_AGE\_THRESHOLD)

{

temp\_task = dummy\_task\_of(temp\_entity);

temp\_task->prio --;

}

}

}

}

* **switched\_from\_dummy**

- Khi một process chuyển từ dummy schedule class sang một class khác, hàm *switched\_from\_dummy* sẽ xóa process này khỏi dummy run queue.

Code:

// When switch from the dummy, we dequeue the task from dummy run queue

static void switched\_from\_dummy(struct rq \*rq, struct task\_struct \*p)

{

dequeue\_task\_dummy(rq, p, 0);

}

* **switched\_to\_dummy**

- Khi một process chuyển từ một class khác sang dummy schedule class, hàm *switched\_to\_dummy* sẽ chèn process mới vào dummy run queue. Nếu process này đang thực thi, ta không làm gì cả, nếu process này chưa được thực thi, chúng ta sẽ kiểm tra xem nó có thể preempt process đang chạy hay không bằng hàm *check\_preempt\_curr\_dummy*.

Code:

// When switch to the dummy, we enqueue the task to dummy run queue

// If the task is running, we do nothing

// If the task is not running, it may preempt the current task.

static void switched\_to\_dummy(struct rq \*rq, struct task\_struct \*p)

{

enqueue\_task\_dummy(rq, p, 0);

if(rq->curr != p)

{

check\_preempt\_curr\_dummy(rq,p,0);

}

}

* **prio\_changed\_dummy**

- Khi độ ưu tiên của một process thay đổi, nếu process này có độ ưu tiên cao hơn process đang chạy, ta sẽ preempt process đang chạy và thay thế bằng process mới.

Code:

/\*

\* When priority of a process is changed

\* If the task has priority higher the current priority, the task will preempt the currently executing task

\*/

static void prio\_changed\_dummy(struct rq \*rq, struct task\_struct \*p, int oldprio)

{

if(p->prio < rq->curr->prio)

{

set\_tsk\_need\_resched(rq->curr);

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

}

}

* **get\_rr\_interval\_dummy**

- Hàm có chức năng lấy *timeslice* được quy định cho một process.

Code:

/\*

\* Get timeslice

\*/

static unsigned int get\_rr\_interval\_dummy(struct rq \*rq, struct task\_struct \*p)

{

return get\_timeslice();

}

**V) Tài liệu tham khảo**

[1] Robert Love, *Linux Kernel Development Third Edition.*

[2] Daniel P. Bovet, Marco Cesati, *Understanding Linux Kernel Third Edition.*

[3] Wolfgang Mauerer, *Professional Linux Kernel Architecture.*

**VI) Phụ lục**

**dummy.c**

/\*

\* Dummy scheduling class, mapped to range of 5 levels of SCHED\_NORMAL policy

\*/

#include "sched.h"

/\*

\* Timeslice and age threshold are represented in jiffies. Default timeslice

\* is 100ms. Both parameters can be tuned from /proc/sys/kernel.

\*/

#define DUMMY\_TIMESLICE (100 \* HZ / 1000)

#define DUMMY\_AGE\_THRESHOLD (3 \* DUMMY\_TIMESLICE)

#define MAX\_DUMMY\_PRIO 135

unsigned int sysctl\_sched\_dummy\_timeslice = DUMMY\_TIMESLICE;

static inline unsigned int get\_timeslice()

{

return sysctl\_sched\_dummy\_timeslice;

}

unsigned int sysctl\_sched\_dummy\_age\_threshold = DUMMY\_AGE\_THRESHOLD;

static inline unsigned int get\_age\_threshold()

{

return sysctl\_sched\_dummy\_age\_threshold;

}

/\*

\* Init

\*/

void init\_dummy\_rq(struct dummy\_rq \*dummy\_rq, struct rq \*rq)

{

INIT\_LIST\_HEAD(&dummy\_rq->queue); // see list.h for more information line 24

}

/\*

\* Helper functions

\*/

static inline struct task\_struct \*dummy\_task\_of(struct sched\_dummy\_entity \*dummy\_se)

{

return container\_of(dummy\_se, struct task\_struct, dummy\_se);

}

/\*

\* Add the task to the tail of dummy run queue and initialize the timeslice and age\_threshold is zero

\*/

static inline void \_enqueue\_task\_dummy(struct rq \*rq, struct task\_struct \*p)

{

struct sched\_dummy\_entity \*dummy\_se = &p->dummy\_se;

struct list\_head \*queue = &rq->dummy.queue;

dummy\_se->timeslice = 0;

dummy\_se->age\_threshold =0;

list\_add\_tail(&dummy\_se->run\_list, queue);

}

/\*

\* Delete a task form the dummy run queue

\*/

static inline void \_dequeue\_task\_dummy(struct task\_struct \*p)

{

struct sched\_dummy\_entity \*dummy\_se = &p->dummy\_se;

list\_del\_init(&dummy\_se->run\_list);

}

/\*

\* Scheduling class functions to implement

\*/

/\*

\* Add the task to the tail of dummy run queue and initialize the timeslice and age\_threshold is zero

\*/

static void enqueue\_task\_dummy(struct rq \*rq, struct task\_struct \*p, int flags)

{

\_enqueue\_task\_dummy(rq, p);

inc\_nr\_running(rq);

}

/\*

\* Delete a task form the dummy run queue

\*/

static void dequeue\_task\_dummy(struct rq \*rq, struct task\_struct \*p, int flags)

{

\_dequeue\_task\_dummy(p);

dec\_nr\_running(rq);

}

/\*

\* When a process wants to relinquish control of the processor voluntarily, it can use the

\* sched\_yield system call. This triggers yield\_task to be called in the kernel.

\*/

static void yield\_task\_dummy(struct rq \*rq)

{

/\*

\* Reset the timeslice and age\_threshold of the current task is zero

\*/

rq->curr->dummy\_se.timeslice = 0;

rq->curr->dummy\_se.age\_threshold = 0;

/\* Delete and Add the current task to the tail of dummy run queue to ensure that

\* the current task will be end of the dummy run queue

\*/

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

}

/\*

\* check\_preempt\_curr is used to preempt the current task with a newly woken task if this is

\* necessary.

\*/

static void check\_preempt\_curr\_dummy(struct rq \*rq, struct task\_struct \*p, int flags)

{

/\*

\* If the task has priority higher than the priority of current task, the currently executing task will be preempted

\* Else, we add the task p to the tail of dummy run queue

\*/

if(p->prio < rq->curr->prio)

{

set\_tsk\_need\_resched(rq->curr);

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

}

else

{

enqueue\_task\_dummy(rq, p, 0);

}

}

//-----------------------------------------------------------------

//-------------------update 30/05----------------------------------

//-----------------------------------------------------------------

/\*

\* Selects the next task that is supposed to run

\* Pick up the highest-prio task:

\*/

static struct task\_struct \*pick\_next\_task\_dummy(struct rq \*rq)

{

/\*

\* Choose the task has highest priority

\*/

struct dummy\_rq \*dummy\_rq = &rq->dummy;

struct list\_head \*temp\_list\_head = NULL;

struct sched\_dummy\_entity \*temp\_entity = NULL;

struct task\_struct \*temp\_task = NULL;

struct task\_struct \*task\_has\_max\_prio = NULL;

int max\_prio;

/\* If dummy run queue not empty, we find the task has highest priority,

\* Else, we return NULL

\*/

if (!list\_empty(&dummy\_rq->queue))

{

temp\_list\_head = dummy\_rq->queue->next;// temp\_list\_head point to the first entry

temp\_entity = list\_entry(temp\_list\_head, struct sched\_dummy\_entity, run\_list);// get the entity that contains

// run\_list = temp\_list\_head

temp\_task = dummy\_task\_of(temp\_entity);// get the task that contains dummy\_se = temp\_entity

task\_has\_max\_prio = temp\_task;

max\_prio = task\_has\_max\_prio->prio;

temp\_list\_head = temp\_list\_head->next;

while(temp\_list\_head != &dummy\_rq->queue)// loop throught all the task and find the task has highest priority

{

temp\_entity = list\_entry(temp\_list\_head, struct sched\_dummy\_entity, run\_list);

temp\_task = dummy\_task\_of(temp\_entity);

if(temp\_task->prio < max\_prio)

{

task\_has\_max\_prio = temp\_task;

max\_prio = task\_has\_mnax\_prio->prio;

}

temp\_list\_head = temp\_list\_head->next;

}

return task\_has\_max\_prio;

}

else

{

return NULL;

}

}

/\*

\* put\_prev\_task is called before

\* the currently executing task is replaced with another one.

\*

\* We delete it from dummy queue and then add it to tail of dummy queue

\*/

static void put\_prev\_task\_dummy(struct rq \*rq, struct task\_struct \*prev)

{

struct sched\_dummy\_entity \*dummy\_se = &rq->curr->dummy\_se; // take entity of the currently executing task

struct list\_head \*queue = &rq->dummy.queue; // take dummy queue

list\_move\_tail(&dummy\_se->run\_list, queue); // see list.h for more information line 164

}

//----------------------------------------------------------------------------------------------------------

/\*

\* set\_curr\_task is called when the scheduling policy of the current task is changed.

\*/

static void set\_curr\_task\_dummy(struct rq \*rq)

{

dequeue\_task\_dummy(rq, rq->curr, 0);

}

/\*

\* task\_tick\_dummy is called by the periodic scheduler each time it is activated.

\*/

static void task\_tick\_dummy(struct rq \*rq, struct task\_struct \*curr, int queued)

{

// Update the current timeslice

curr->dummy\_se->timeslice++;

// If the current task exhaust the timeslice, the currently executing task can be replaced with another one.

if(curr->dummy\_se->timeslice == DUMMY\_TIMESLICE)

{

curr->dummy\_se->timeslice = 0;

set\_tsk\_need\_resched(curr);

// We dequeue and enqueue the current task to ensure that the current task is moved to the tail of queue

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

return;

}

// Update the age\_threshold of all the waiting task

struct dummy\_rq \*dummy\_rq = &rq->dummy;

struct list\_head \*temp\_list\_head = dummy\_rq->queue.next;

struct sched\_dummy\_entity \*temp\_entity = NULL;

struct task\_struct \*temp\_task;

if (!list\_empty(&dummy\_rq->queue))

{

// Loop through all the task and update the age\_threshold

// If the age\_threshold equal DUMMY\_AGE\_THRESHOLD, we increase the priority by 1

while(temp\_list\_head != &dummy\_rq->queue)

{

temp\_entity = list\_entry(temp\_list\_head, struct sched\_dummy\_entity, run\_list);

temp\_entity->age\_threshold++;

if(temp\_entity->age\_threshold == DUMMY\_AGE\_THRESHOLD)

{

temp\_task = dummy\_task\_of(temp\_entity);

temp\_task->prio --;

}

}

}

}

// When switch from the dummy, we dequeue the task from dummy run queue

static void switched\_from\_dummy(struct rq \*rq, struct task\_struct \*p)

{

dequeue\_task\_dummy(rq, p, 0);

}

// When switch to the dummy, we enqueue the task to dummy run queue

// If the task is running, we do nothing

// If the task is not running, it may preempt the current task.

static void switched\_to\_dummy(struct rq \*rq, struct task\_struct \*p)

{

enqueue\_task\_dummy(rq, p, 0);

if(rq->curr != p)

{

check\_preempt\_curr\_dummy(rq,p,0);

}

}

/\*

\* When priority of a process is changed

\* If the task has priority higher the current priority, the task will preempt the currently executing task

\*/

static void prio\_changed\_dummy(struct rq \*rq, struct task\_struct \*p, int oldprio)

{

if(p->prio < rq->curr->prio)

{

set\_tsk\_need\_resched(rq->curr);

dequeue\_task\_dummy(rq, rq->curr, 0);

enqueue\_task\_dummy(rq, rq->curr, 0);

}

}

/\*

\* Get timeslice

\*/

static unsigned int get\_rr\_interval\_dummy(struct rq \*rq, struct task\_struct \*p)

{

return get\_timeslice();

}

/\*

\* Scheduling class

\*/

const struct sched\_class dummy\_sched\_class = {

.next = &idle\_sched\_class,

.enqueue\_task = enqueue\_task\_dummy,

.dequeue\_task = dequeue\_task\_dummy,

.yield\_task = yield\_task\_dummy,

.check\_preempt\_curr = check\_preempt\_curr\_dummy,

.pick\_next\_task = pick\_next\_task\_dummy,

.put\_prev\_task = put\_prev\_task\_dummy,

.set\_curr\_task = set\_curr\_task\_dummy,

.task\_tick = task\_tick\_dummy,

.switched\_from = switched\_from\_dummy,

.switched\_to = switched\_to\_dummy,

.prio\_changed = prio\_changed\_dummy,

.get\_rr\_interval = get\_rr\_interval\_dummy

};

**struct task\_struct**

struct task\_struct {

volatile long state; /\* -1 unrunnable, 0 runnable, >0 stopped \*/

void \*stack;

atomic\_t usage;

unsigned int flags; /\* per process flags, defined below \*/

unsigned int ptrace;

#ifdef CONFIG\_SMP

struct llist\_node wake\_entry;

int on\_cpu;

#endif

int on\_rq;

int prio, static\_prio, normal\_prio;

unsigned int rt\_priority;

const struct sched\_class \*sched\_class;

struct sched\_entity se;

struct sched\_rt\_entity rt;

struct sched\_dummy\_entity dummy\_se;

#ifdef CONFIG\_CGROUP\_SCHED

struct task\_group \*sched\_task\_group;

#endif

#ifdef CONFIG\_PREEMPT\_NOTIFIERS

/\* list of struct preempt\_notifier: \*/

struct hlist\_head preempt\_notifiers;

#endif

/\*

\* fpu\_counter contains the number of consecutive context switches

\* that the FPU is used. If this is over a threshold, the lazy fpu

\* saving becomes unlazy to save the trap. This is an unsigned char

\* so that after 256 times the counter wraps and the behavior turns

\* lazy again; this to deal with bursty apps that only use FPU for

\* a short time

\*/

unsigned char fpu\_counter;

#ifdef CONFIG\_BLK\_DEV\_IO\_TRACE

unsigned int btrace\_seq;

#endif

unsigned int policy;

int nr\_cpus\_allowed;

cpumask\_t cpus\_allowed;

#ifdef CONFIG\_PREEMPT\_RCU

int rcu\_read\_lock\_nesting;

char rcu\_read\_unlock\_special;

struct list\_head rcu\_node\_entry;

#endif /\* #ifdef CONFIG\_PREEMPT\_RCU \*/

#ifdef CONFIG\_TREE\_PREEMPT\_RCU

struct rcu\_node \*rcu\_blocked\_node;

#endif /\* #ifdef CONFIG\_TREE\_PREEMPT\_RCU \*/

#ifdef CONFIG\_RCU\_BOOST

struct rt\_mutex \*rcu\_boost\_mutex;

#endif /\* #ifdef CONFIG\_RCU\_BOOST \*/

#if defined(CONFIG\_SCHEDSTATS) || defined(CONFIG\_TASK\_DELAY\_ACCT)

struct sched\_info sched\_info;

#endif

struct list\_head tasks;

#ifdef CONFIG\_SMP

struct plist\_node pushable\_tasks;

#endif

struct mm\_struct \*mm, \*active\_mm;

#ifdef CONFIG\_COMPAT\_BRK

unsigned brk\_randomized:1;

#endif

#if defined(SPLIT\_RSS\_COUNTING)

struct task\_rss\_stat rss\_stat;

#endif

/\* task state \*/

int exit\_state;

int exit\_code, exit\_signal;

int pdeath\_signal; /\* The signal sent when the parent dies \*/

unsigned int jobctl; /\* JOBCTL\_\*, siglock protected \*/

/\* Used for emulating ABI behavior of previous Linux versions \*/

unsigned int personality;

unsigned did\_exec:1;

unsigned in\_execve:1; /\* Tell the LSMs that the process is doing an

\* execve \*/

unsigned in\_iowait:1;

/\* task may not gain privileges \*/

unsigned no\_new\_privs:1;

/\* Revert to default priority/policy when forking \*/

unsigned sched\_reset\_on\_fork:1;

unsigned sched\_contributes\_to\_load:1;

pid\_t pid;

pid\_t tgid;

#ifdef CONFIG\_CC\_STACKPROTECTOR

/\* Canary value for the -fstack-protector gcc feature \*/

unsigned long stack\_canary;

#endif

/\*

\* pointers to (original) parent process, youngest child, younger sibling,

\* older sibling, respectively. (p->father can be replaced with

\* p->real\_parent->pid)

\*/

struct task\_struct \_\_rcu \*real\_parent; /\* real parent process \*/

struct task\_struct \_\_rcu \*parent; /\* recipient of SIGCHLD, wait4() reports \*/

/\*

\* children/sibling forms the list of my natural children

\*/

struct list\_head children; /\* list of my children \*/

struct list\_head sibling; /\* linkage in my parent's children list \*/

struct task\_struct \*group\_leader; /\* threadgroup leader \*/

/\*

\* ptraced is the list of tasks this task is using ptrace on.

\* This includes both natural children and PTRACE\_ATTACH targets.

\* p->ptrace\_entry is p's link on the p->parent->ptraced list.

\*/

struct list\_head ptraced;

struct list\_head ptrace\_entry;

/\* PID/PID hash table linkage. \*/

struct pid\_link pids[PIDTYPE\_MAX];

struct list\_head thread\_group;

struct completion \*vfork\_done; /\* for vfork() \*/

int \_\_user \*set\_child\_tid; /\* CLONE\_CHILD\_SETTID \*/

int \_\_user \*clear\_child\_tid; /\* CLONE\_CHILD\_CLEARTID \*/

cputime\_t utime, stime, utimescaled, stimescaled;

cputime\_t gtime;

#ifndef CONFIG\_VIRT\_CPU\_ACCOUNTING\_NATIVE

struct cputime prev\_cputime;

#endif

#ifdef CONFIG\_VIRT\_CPU\_ACCOUNTING\_GEN

seqlock\_t vtime\_seqlock;

unsigned long long vtime\_snap;

enum {

VTIME\_SLEEPING = 0,

VTIME\_USER,

VTIME\_SYS,

} vtime\_snap\_whence;

#endif

unsigned long nvcsw, nivcsw; /\* context switch counts \*/

struct timespec start\_time; /\* monotonic time \*/

struct timespec real\_start\_time; /\* boot based time \*/

/\* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific \*/

unsigned long min\_flt, maj\_flt;

struct task\_cputime cputime\_expires;

struct list\_head cpu\_timers[3];

/\* process credentials \*/

const struct cred \_\_rcu \*real\_cred; /\* objective and real subjective task

\* credentials (COW) \*/

const struct cred \_\_rcu \*cred; /\* effective (overridable) subjective task

\* credentials (COW) \*/

char comm[TASK\_COMM\_LEN]; /\* executable name excluding path

- access with [gs]et\_task\_comm (which lock

it with task\_lock())

- initialized normally by setup\_new\_exec \*/

/\* file system info \*/

int link\_count, total\_link\_count;

#ifdef CONFIG\_SYSVIPC

/\* ipc stuff \*/

struct sysv\_sem sysvsem;

#endif

#ifdef CONFIG\_DETECT\_HUNG\_TASK

/\* hung task detection \*/

unsigned long last\_switch\_count;

#endif

/\* CPU-specific state of this task \*/

struct thread\_struct thread;

/\* filesystem information \*/

struct fs\_struct \*fs;

/\* open file information \*/

struct files\_struct \*files;

/\* namespaces \*/

struct nsproxy \*nsproxy;

/\* signal handlers \*/

struct signal\_struct \*signal;

struct sighand\_struct \*sighand;

sigset\_t blocked, real\_blocked;

sigset\_t saved\_sigmask; /\* restored if set\_restore\_sigmask() was used \*/

struct sigpending pending;

unsigned long sas\_ss\_sp;

size\_t sas\_ss\_size;

int (\*notifier)(void \*priv);

void \*notifier\_data;

sigset\_t \*notifier\_mask;

struct callback\_head \*task\_works;

struct audit\_context \*audit\_context;

#ifdef CONFIG\_AUDITSYSCALL

kuid\_t loginuid;

unsigned int sessionid;

#endif

struct seccomp seccomp;

/\* Thread group tracking \*/

u32 parent\_exec\_id;

u32 self\_exec\_id;

/\* Protection of (de-)allocation: mm, files, fs, tty, keyrings, mems\_allowed,

\* mempolicy \*/

spinlock\_t alloc\_lock;

/\* Protection of the PI data structures: \*/

raw\_spinlock\_t pi\_lock;

#ifdef CONFIG\_RT\_MUTEXES

/\* PI waiters blocked on a rt\_mutex held by this task \*/

struct plist\_head pi\_waiters;

/\* Deadlock detection and priority inheritance handling \*/

struct rt\_mutex\_waiter \*pi\_blocked\_on;

#endif

#ifdef CONFIG\_DEBUG\_MUTEXES

/\* mutex deadlock detection \*/

struct mutex\_waiter \*blocked\_on;

#endif

#ifdef CONFIG\_TRACE\_IRQFLAGS

unsigned int irq\_events;

unsigned long hardirq\_enable\_ip;

unsigned long hardirq\_disable\_ip;

unsigned int hardirq\_enable\_event;

unsigned int hardirq\_disable\_event;

int hardirqs\_enabled;

int hardirq\_context;

unsigned long softirq\_disable\_ip;

unsigned long softirq\_enable\_ip;

unsigned int softirq\_disable\_event;

unsigned int softirq\_enable\_event;

int softirqs\_enabled;

int softirq\_context;

#endif

#ifdef CONFIG\_LOCKDEP

# define MAX\_LOCK\_DEPTH 48UL

u64 curr\_chain\_key;

int lockdep\_depth;

unsigned int lockdep\_recursion;

struct held\_lock held\_locks[MAX\_LOCK\_DEPTH];

gfp\_t lockdep\_reclaim\_gfp;

#endif

/\* journalling filesystem info \*/

void \*journal\_info;

/\* stacked block device info \*/

struct bio\_list \*bio\_list;

#ifdef CONFIG\_BLOCK

/\* stack plugging \*/

struct blk\_plug \*plug;

#endif

/\* VM state \*/

struct reclaim\_state \*reclaim\_state;

struct backing\_dev\_info \*backing\_dev\_info;

struct io\_context \*io\_context;

unsigned long ptrace\_message;

siginfo\_t \*last\_siginfo; /\* For ptrace use. \*/

struct task\_io\_accounting ioac;

#if defined(CONFIG\_TASK\_XACCT)

u64 acct\_rss\_mem1; /\* accumulated rss usage \*/

u64 acct\_vm\_mem1; /\* accumulated virtual memory usage \*/

cputime\_t acct\_timexpd; /\* stime + utime since last update \*/

#endif

#ifdef CONFIG\_CPUSETS

nodemask\_t mems\_allowed; /\* Protected by alloc\_lock \*/

seqcount\_t mems\_allowed\_seq; /\* Seqence no to catch updates \*/

int cpuset\_mem\_spread\_rotor;

int cpuset\_slab\_spread\_rotor;

#endif

#ifdef CONFIG\_CGROUPS

/\* Control Group info protected by css\_set\_lock \*/

struct css\_set \_\_rcu \*cgroups;

/\* cg\_list protected by css\_set\_lock and tsk->alloc\_lock \*/

struct list\_head cg\_list;

#endif

#ifdef CONFIG\_FUTEX

struct robust\_list\_head \_\_user \*robust\_list;

#ifdef CONFIG\_COMPAT

struct compat\_robust\_list\_head \_\_user \*compat\_robust\_list;

#endif

struct list\_head pi\_state\_list;

struct futex\_pi\_state \*pi\_state\_cache;

#endif

#ifdef CONFIG\_PERF\_EVENTS

struct perf\_event\_context \*perf\_event\_ctxp[perf\_nr\_task\_contexts];

struct mutex perf\_event\_mutex;

struct list\_head perf\_event\_list;

#endif

#ifdef CONFIG\_NUMA

struct mempolicy \*mempolicy; /\* Protected by alloc\_lock \*/

short il\_next;

short pref\_node\_fork;

#endif

#ifdef CONFIG\_NUMA\_BALANCING

int numa\_scan\_seq;

int numa\_migrate\_seq;

unsigned int numa\_scan\_period;

u64 node\_stamp; /\* migration stamp \*/

struct callback\_head numa\_work;

#endif /\* CONFIG\_NUMA\_BALANCING \*/

struct rcu\_head rcu;

/\*

\* cache last used pipe for splice

\*/

struct pipe\_inode\_info \*splice\_pipe;

struct page\_frag task\_frag;

#ifdef CONFIG\_TASK\_DELAY\_ACCT

struct task\_delay\_info \*delays;

#endif

#ifdef CONFIG\_FAULT\_INJECTION

int make\_it\_fail;

#endif

/\*

\* when (nr\_dirtied >= nr\_dirtied\_pause), it's time to call

\* balance\_dirty\_pages() for some dirty throttling pause

\*/

int nr\_dirtied;

int nr\_dirtied\_pause;

unsigned long dirty\_paused\_when; /\* start of a write-and-pause period \*/

#ifdef CONFIG\_LATENCYTOP

int latency\_record\_count;

struct latency\_record latency\_record[LT\_SAVECOUNT];

#endif

/\*

\* time slack values; these are used to round up poll() and

\* select() etc timeout values. These are in nanoseconds.

\*/

unsigned long timer\_slack\_ns;

unsigned long default\_timer\_slack\_ns;

#ifdef CONFIG\_FUNCTION\_GRAPH\_TRACER

/\* Index of current stored address in ret\_stack \*/

int curr\_ret\_stack;

/\* Stack of return addresses for return function tracing \*/

struct ftrace\_ret\_stack \*ret\_stack;

/\* time stamp for last schedule \*/

unsigned long long ftrace\_timestamp;

/\*

\* Number of functions that haven't been traced

\* because of depth overrun.

\*/

atomic\_t trace\_overrun;

/\* Pause for the tracing \*/

atomic\_t tracing\_graph\_pause;

#endif

#ifdef CONFIG\_TRACING

/\* state flags for use by tracers \*/

unsigned long trace;

/\* bitmask and counter of trace recursion \*/

unsigned long trace\_recursion;

#endif /\* CONFIG\_TRACING \*/

#ifdef CONFIG\_MEMCG /\* memcg uses this to do batch job \*/

struct memcg\_batch\_info {

int do\_batch; /\* incremented when batch uncharge started \*/

struct mem\_cgroup \*memcg; /\* target memcg of uncharge \*/

unsigned long nr\_pages; /\* uncharged usage \*/

unsigned long memsw\_nr\_pages; /\* uncharged mem+swap usage \*/

} memcg\_batch;

unsigned int memcg\_kmem\_skip\_account;

#endif

#ifdef CONFIG\_UPROBES

struct uprobe\_task \*utask;

#endif

#if defined(CONFIG\_BCACHE) || defined(CONFIG\_BCACHE\_MODULE)

unsigned int sequential\_io;

unsigned int sequential\_io\_avg;

#endif

};

**struct rq**

struct rq {

/\* runqueue lock: \*/

raw\_spinlock\_t lock;

/\*

\* nr\_running and cpu\_load should be in the same cacheline because

\* remote CPUs use both these fields when doing load calculation.

\*/

unsigned int nr\_running;

#define CPU\_LOAD\_IDX\_MAX 5

unsigned long cpu\_load[CPU\_LOAD\_IDX\_MAX];

unsigned long last\_load\_update\_tick;

#ifdef CONFIG\_NO\_HZ\_COMMON

u64 nohz\_stamp;

unsigned long nohz\_flags;

#endif

#ifdef CONFIG\_NO\_HZ\_FULL

unsigned long last\_sched\_tick;

#endif

int skip\_clock\_update;

/\* capture load from \*all\* tasks on this cpu: \*/

struct load\_weight load;

unsigned long nr\_load\_updates;

u64 nr\_switches;

struct cfs\_rq cfs;

struct rt\_rq rt;

struct dummy\_rq dummy;

#ifdef CONFIG\_FAIR\_GROUP\_SCHED

/\* list of leaf cfs\_rq on this cpu: \*/

struct list\_head leaf\_cfs\_rq\_list;

#ifdef CONFIG\_SMP

unsigned long h\_load\_throttle;

#endif /\* CONFIG\_SMP \*/

#endif /\* CONFIG\_FAIR\_GROUP\_SCHED \*/

#ifdef CONFIG\_RT\_GROUP\_SCHED

struct list\_head leaf\_rt\_rq\_list;

#endif

/\*

\* This is part of a global counter where only the total sum

\* over all CPUs matters. A task can increase this counter on

\* one CPU and if it got migrated afterwards it may decrease

\* it on another CPU. Always updated under the runqueue lock:

\*/

unsigned long nr\_uninterruptible;

struct task\_struct \*curr, \*idle, \*stop;

unsigned long next\_balance;

struct mm\_struct \*prev\_mm;

u64 clock;

u64 clock\_task;

atomic\_t nr\_iowait;

#ifdef CONFIG\_SMP

struct root\_domain \*rd;

struct sched\_domain \*sd;

unsigned long cpu\_power;

unsigned char idle\_balance;

/\* For active balancing \*/

int post\_schedule;

int active\_balance;

int push\_cpu;

struct cpu\_stop\_work active\_balance\_work;

/\* cpu of this runqueue: \*/

int cpu;

int online;

struct list\_head cfs\_tasks;

u64 rt\_avg;

u64 age\_stamp;

u64 idle\_stamp;

u64 avg\_idle;

#endif

#ifdef CONFIG\_IRQ\_TIME\_ACCOUNTING

u64 prev\_irq\_time;

#endif

#ifdef CONFIG\_PARAVIRT

u64 prev\_steal\_time;

#endif

#ifdef CONFIG\_PARAVIRT\_TIME\_ACCOUNTING

u64 prev\_steal\_time\_rq;

#endif

/\* calc\_load related fields \*/

unsigned long calc\_load\_update;

long calc\_load\_active;

#ifdef CONFIG\_SCHED\_HRTICK

#ifdef CONFIG\_SMP

int hrtick\_csd\_pending;

struct call\_single\_data hrtick\_csd;

#endif

struct hrtimer hrtick\_timer;

#endif

#ifdef CONFIG\_SCHEDSTATS

/\* latency stats \*/

struct sched\_info rq\_sched\_info;

unsigned long long rq\_cpu\_time;

/\* could above be rq->cfs\_rq.exec\_clock + rq->rt\_rq.rt\_runtime ? \*/

/\* sys\_sched\_yield() stats \*/

unsigned int yld\_count;

/\* schedule() stats \*/

unsigned int sched\_count;

unsigned int sched\_goidle;

/\* try\_to\_wake\_up() stats \*/

unsigned int ttwu\_count;

unsigned int ttwu\_local;

#endif

#ifdef CONFIG\_SMP

struct llist\_head wake\_list;

#endif

struct sched\_avg avg;

};

static inline int cpu\_of(struct rq \*rq)

{

#ifdef CONFIG\_SMP

return rq->cpu;

#else

return 0;

#endif

}