# 调度器

## 调度器基本架构

### task\_structs

task\_struct 是对进程的抽象, 每一个进程/线程就抽象为一个 task. 这里面有一些关于进程的信息. 和调度最相关的就是下面这些了. on\_rq 表示是否在运行队列上.

/\* include/linux/sched.h \*/

struct task\_struct {

/\* -1 unrunnable, 0 runnable, >0 stopped: \*/

volatile long state;

/\* 栈指针 \*/

void \*stack;

/\* 是否在就绪队列上 \*/

int on\_rq;

/\* 优先级 \*/

int prio;

int static\_prio;

int normal\_prio;

unsigned int rt\_priority;

/\* 调度器相关 \*/

const struct sched\_class \*sched\_class;

struct sched\_entity se;

struct sched\_rt\_entity rt;

struct sched\_dl\_entity dl;

/\* 调度策略 \*/

unsigned int policy;

/\* 省略几百行和调度不那么相关的 \*/

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

struct thread\_struct thread;

};

### RQ

每个 CPU 会维护一个就绪队列 runqueue . 同时还会持有三个 rq 供不同的调度器使用.

/\* kernel/sched/sched.h \*/

/\*

\* This is the main, per-CPU runqueue data structure.

\*

\* Locking rule: those places that want to lock multiple runqueues

\* (such as the load balancing or the thread migration code), lock

\* acquire operations must be ordered by ascending &runqueue.

\*/

struct rq {

/\* 锁 \*/

raw\_spinlock\_t lock;

/\* 运行中的进程数 \*/

unsigned int nr\_running;

/\* 三个不同的 rq, 供三个调度器分别使用 \*/

struct cfs\_rq cfs;

struct rt\_rq rt;

struct dl\_rq dl;

/\* 指向任务的指针 \*/

struct task\_struct \_\_rcu \*curr;

struct task\_struct \*idle;

struct task\_struct \*stop;

/\* 省略其余 \*/

};

### 进程状态

进程描述符的state字段用于保存进程的当前状态， 进程的状态有以下几种：

TASK\_RUNNING (运行)– 进程处于可执行状态， 在这个状态下的进程要么正在被CPU执行， 要么在等待执行（CPU被其他进程占用的情况下）.

TASK\_INTERRUPTIBLE (可中断等待)– 进程处于等待状态, 其在等待某些条件成立或者接收到某些信号, 进程会被唤醒变为运行状态.

TASK\_UNINTERRUPTIBLE (不可中断等待)– 进程处于等待状态, 其在等待某些条件成立, 进程会被唤醒变为运行状态, 但不能被信号唤醒.

TASK\_TRACED (被追踪)– 进程处于被追踪状态， 例如通过ptrace命令对进程进行调试.

TASK\_STOPPED (停止)– 进程处于停止状态, 进程不能被执行.一般接收到SIGSTOP， SIGTSTP， SIGTTIN， SIGTTOU信号进程会变成TASK\_STOPPED状态.

### 调度器类和调度策略

调度器类实际上就是一族暴露给主调度器的 API

调度器类有(优先级从高到低)

\_\_stop\_sched\_class

\_\_dl\_sched\_class

\_\_rt\_sched\_class

\_\_fair\_sched\_class

\_\_idle\_sched\_class

enqueue\_task(…): 一个 task 变成就绪状态, 希望挂上这个调度器的就绪队列(rq).

dequeue\_task(…): 一个 task 不再就绪了(比如阻塞了), 要从调度器的就绪队列上离开.

yield\_task(…): 跳过当前任务.

yield\_to\_task(p): 跳过当前任务, 并且尽量调度任务 p.

check\_preempt\_curr(…): 这个函数在有任务被唤醒时候调用, 看看能不能抢占当前任务.

pick\_next\_task(…): 选择下一个就绪的任务以运行.

task\_tick(…): 大部分情况下是用作时钟中断的回调.

Linux 内核中实现了多个调度算法, 每一个算法由不同的调度器类 sched\_class 来表示. 其实它就是一族 API. 每个调度器实现了这些 API, 就可以供上层 (也就是主调度器) 使用了.

/\* kernel/sched/sched.h \*/

struct sched\_class {

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);

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);

void (\*set\_next\_task)(struct rq \*rq, struct task\_struct \*p, bool first);

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

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

void (\*task\_dead)(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);

void (\*update\_curr)(struct rq \*rq);

};

### 调度策略

调度策略有(优先级从高到低)

/\* include/uapi/linux/sched.h \*/

/\*

\* Scheduling policies

\*/

#define SCHED\_NORMAL 0

#define SCHED\_FIFO 1

#define SCHED\_RR 2

#define SCHED\_BATCH 3

#define SCHED\_IDLE 5

#define SCHED\_DEADLINE 6

SCHED\_DEADLINE

SCHED\_FIFO

SCHED\_RR

SCHED\_NORMAL

SCHED\_BATCH

SCHED\_IDLE

### 调度实体 sched\_entity

为了调度 task\_struct , 最简单的想法是搞一些 task\_struct\* . 但这样其实不够好, 因为调度总是需要一些比 task\_struct\* 更详细的信息, 比如当前运行时间或者上次运行时间. 所以我们就搞出了调度实体, 叫做 sched\_entity , 储存有一些调度相关的信息, 并且专门用来给调度器调度.

我们在最上面的代码中可以看到, 每个 task\_struct 都有一个指针, 指向自己的调度器. 并且它为 DL, RT, CFS 这三个调度器都预留了调度实体. 最常用的就是 CFS 调度器了.

/\* include/linux/sched.h \*/

struct task\_struct {

/\* 调度器相关 \*/

const struct sched\_class \*sched\_class;

struct sched\_entity se; /\* 用来给 CFS 调度器调度 \*/

struct sched\_rt\_entity rt; /\* 用来给 RT 调度器调度 \*/

struct sched\_dl\_entity dl; /\* 用来给 DL 调度器调度 \*/

};

可以想象, 这些 sched\_entity 类型里面存有此类调度器需要的信息.

/\* include/linux/sched.h \*/

struct sched\_entity {

/\* For load-balancing: \*/

struct load\_weight load;

struct rb\_node run\_node;

struct list\_head group\_node;

unsigned int on\_rq;

u64 exec\_start;

u64 sum\_exec\_runtime;

u64 vruntime;

u64 prev\_sum\_exec\_runtime;

u64 nr\_migrations;

struct sched\_statistics statistics;

};

如果需要从 sched\_entity 回到 task\_struct\* , 则只需要减去一个成员的偏移量即可, 非常方便 (且危险).

/\* include/linux/kernel.h \*/

#define container\_of(ptr, type, member) ({ \

void \*\_\_mptr = (void \*)(ptr); \

BUILD\_BUG\_ON\_MSG(!\_\_same\_type(\*(ptr), ((type \*)0)->member) && \

!\_\_same\_type(\*(ptr), void), \

"pointer type mismatch in container\_of()"); \

((type \*)(\_\_mptr - offsetof(type, member))); })

/\* kernel/sched/fair.c \*/

static inline struct task\_struct \*task\_of(struct sched\_entity \*se)

{

return container\_of(se, struct task\_struct, se);

}

### 静态优先级

在创建进程时确定的，且程序的整个运行期间不变

范围为100-139(MAX\_RT\_PRIO - MAX\_PRIO-1)

内核2.6中的静态优先级相当于内核2.4中的nice值，但转到MAX\_RT\_PRIO到MAX\_PRIO-1取值范围，其公式为：

static priority = nice + 20 + MAX\_RT\_PRIO

内核定义两个宏来实现此转化：nice\_to\_prio() and prio\_to\_nice()

#define NICE\_TO\_PRIO(nice) (MAX\_RT\_PRIO + (nice) + 20)

#define PRIO\_TO\_NICE(prio) ((prio) - MAX\_RT\_PRIO - 20)

### 归一化优先级（normalized priority）

task struct中的normal\_prio成员。我们称之归一化优先级（normalized priority），它是根据静态优先级、scheduling priority和调度策略来计算得到，代码如下：

static inline int normal\_prio(struct task\_struct \*p)

{

int prio;

if (task\_has\_dl\_policy(p))

prio = MAX\_DL\_PRIO-1;

else if (task\_has\_rt\_policy(p))

prio = MAX\_RT\_PRIO-1 - p->rt\_priority;

else

prio = \_\_normal\_prio(p);

return prio;

}

对于这里的优先级，调度器需要综合考虑各种因素，例如调度策略，nice value、scheduling priority等，把这些factor全部考虑进来，归一化成一个数轴上的number，以此来表示其优先级，这就是normalized priority。对于一个线程，其normalized priority的number越小，其优先级越大。

调度策略是deadline的进程比RT进程和normal进程的优先级还要高，因此它的归一化优先级是负数：-1。如果采用实时调度策略，那么该线程的normalized priority和rt\_priority相关。task struct中的rt\_priority成员是用户空间视角的实时优先级（scheduling priority），MAX\_RT\_PRIO-1是99，MAX\_RT\_PRIO-1 - p->rt\_priority则翻转了实时进程的scheduling priority，最高优先级是0，最低是98。顺便说一句，normalized priority是99的情况是没有意义的。对于普通进程，normalized priority就是其静态优先级

### 动态优先级

在程序运行期间可根据进程推进或随其等待时间的增加而改变

范围与静态优先级相同。

动态优先级是调度器选择一个进程时实际参考的值。

dynamic priority = max( 100, min((static priority - bonus + 5), 139))

即动态优先级与静态优先级和bonus值有关，而bonus取值范围为0-10，其值大小与平均睡眠时间（average sleeping time）有关。

平均睡眠时间越大，bonus值越大，其动态优先级越大。（睡眠时间长了自然要提高其动态优先级以避免发生饥饿现象）

在task\_struct中用int prio来表示。

动态优先级的计算过程在各个进程的运行过程中进行，同时影响动态优先级大小的因素集中反映在sleep\_avg上。

task\_struct中的成员 unsigned long sleep\_avg;

进程创建时，在wake\_up\_forked\_process()中子进程继承了父进程的动态优先级，并添加到父进程的就绪队列中，如果父进程不在任何就绪队列中，则通过effective\_prio()来计算出子进程的动态优先级并根据计算结果将子进程放置到相应的就绪队列中。

动态优先级的计算只要由effective\_prio()函数来完成：

static int effective\_prio(task\_t \*p)

{

int bonus, prio;

if (rt\_task(p));;如果是实时进程则返回（实时进程的动态优先级是由setscheduler()函数设置且一经设置不再改变）

return p->prio;

bonus = CURRENT\_BONUS(p) - MAX\_BONUS / 2; ;;根据睡眠时间设置bonus值

prio = p->static\_prio - bonus; ;;由静态优先级和bonus值来设置动态优先级

if (prio < MAX\_RT\_PRIO);;动态优先级范围不能小于MAX\_RT\_PRIO也不能超过MAX\_PRIO，所以此处做下判断。

prio = MAX\_RT\_PRIO;

if (prio > MAX\_PRIO-1)

prio = MAX\_PRIO-1;

return prio;

}

#define CURRENT\_BONUS(p) \

(NS\_TO\_JIFFIES((p)->sleep\_avg) \* MAX\_BONUS / \ ;;把睡眠时间转化为bonus值

MAX\_SLEEP\_AVG)

最大bonus值的计算如下：

#define USER\_PRIO(p) ((p)-MAX\_RT\_PRIO)

#define MAX\_USER\_PRIO (USER\_PRIO(MAX\_PRIO)) ;;40

#define PRIO\_BONUS\_RATIO 25

#define MAX\_BONUS (MAX\_USER\_PRIO \* PRIO\_BONUS\_RATIO / 100) ;;结果为10

最大平均睡眠时间计算如下：

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

#define MAX\_SLEEP\_AVG (DEF\_TIMESLICE \* MAX\_BONUS)

### 实时优先级

静态优先级与非实时进程相同

动态优先级在setscheduler()中设置的且一经设置便不再改变

## CFS调度

### 虚拟时间

给每一个任务一个虚拟运行时 (vruntime), 每次挑选 vruntime 最小的任务上去执行. 虚拟运行时是一个理想时间, 通过真实运行时间和权重进行计算. 可以看到这个函数负责更新当前运行任务的 vruntime.

/\* kernel/sched/fair.c \*/

static void update\_curr(struct cfs\_rq \*cfs\_rq)

{

struct sched\_entity \*curr = cfs\_rq->curr;

/\* 省略 \*/

curr->sum\_exec\_runtime += delta\_exec;

schedstat\_add(cfs\_rq->exec\_clock, delta\_exec);

curr->vruntime += calc\_delta\_fair(delta\_exec, curr);

update\_min\_vruntime(cfs\_rq);

/\* 省略 \*/

}

虚拟运行时是真实运行时间除以权重, 权重越大虚拟运行时增长越慢, 于是就能运行更久. 当然, 对于一般的任务而言, 这段虚拟运行时就等于真实运行时间. (权重等于默认值)

对于这里的 vruntime 的具体计算, 可以参考函数

/\* kernel/sched/fair.c \*/

static inline u64 calc\_delta\_fair(u64 delta, struct sched\_entity \*se);

那我们还熟悉一个名词, 叫做任务的优先级. 优先级到权重是有一个转换关系的. 通过这个常数数组进行转换.

/\* kernel/sched/core.c \*/

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,

};

权重就记录在我们上篇文章在 sched\_entity 里面, 的 load 里面.

/\* include/linux/sched.h \*/

struct sched\_entity {

struct load\_weight load;

struct rb\_node run\_node;

u64 sum\_exec\_runtime;

u64 vruntime;

/\* 省略 \*/

};

### 红黑树

红黑树的性质：

1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些约束强制了红黑树的关键性质: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

平衡二叉树的性质：

它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

区别：

1、红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

2、平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知

我们需要一个高效的数据结构来组织这些 sched\_entity . 我们采用的是红黑树, 每个 sched\_entity 贡献出一个 rb\_node 作为红黑树的节点. 红黑树的实现是交由 Linux 中别的模块完成的, CFS 调度器这里只需要实现比较大小就行了.

/\* kernel/sched/sched.h \*/

/\* CFS-related fields in a runqueue \*/

struct cfs\_rq {

struct load\_weight load;

unsigned int nr\_running;

unsigned int h\_nr\_running; /\* SCHED\_{NORMAL,BATCH,IDLE} \*/

unsigned int idle\_h\_nr\_running; /\* SCHED\_IDLE \*/

u64 exec\_clock;

u64 min\_vruntime;

struct rb\_root\_cached tasks\_timeline;

/\*

\* 'curr' points to currently running entity on this cfs\_rq.

\* It is set to NULL otherwise (i.e when none are currently running).

\*/

struct sched\_entity \*curr;

struct sched\_entity \*next;

struct sched\_entity \*last;

struct sched\_entity \*skip;

/\* 省略 \*/

};

所以每个 sched\_entity 都贡献出了 rb\_node , 然后挂上这棵红黑树 tasks\_timeline . 这里的 tasks\_timeline 代表的就是就绪队列.

这个地方说明一下, 红黑树的头文件和实现可以在下面两个地方找到.

include/linux/rbtree.h

lib/rbtree.c

而且这个红黑树节点内部只有父节点颜色 + 两个子节点指针. 怎么让每个红黑树节点保存一个值都是由调用者处理.

我们采用的是红黑树版本会缓存最左侧的节点, 所以查找最左侧节点的效率是 O(1) 的.

可以想象, 随着最左端任务运行, 其 vruntime 增加, 就会被挂到红黑树靠右端. 而右侧的红黑树节点也会慢慢地挪动到左端, 得以运行.

### 调度周期

### 组调度

希望不同分组的任务在高负载下能分配可控比例的CPU资源。为什么会有这个需求呢，比如多用户计算机系统每个用户的所有任务划分到一个分组中，A用户90个相同任务，而B用户只有10个相同任务，在CPU完全跑满的情况下，那么A用户将占90%的CPU时间，而B用户只占到了10%的CPU时间，这对B用户显然是不公平的。再或者同一个用户，既想-j64快速编译，又不想被编译任务影响使用体验，也可将编译任务设置到对应分组中，限制其CPU资源

/dev/cpuctl # ls -l

drwxr-xr-x background

drwxr-xr-x foreground

drwxr-xr-x top-app

...

-rw-r--r-- cpu.shares

-rw-rw-rw- tasks

-rw-rw-rw- cgroup.procs

## 负载跟踪

### PELT

为了实现sched entity级别的负载跟踪，pelt将物理时间划分成了1ms（实际为1024us）的序列，在每一个1024us的周期中，sched entity对系统负载的贡献可以根据该entity处于runnable状态（包含在cpu上running和在cfs\_rq上waiting的状态）的时间进行计算；假设该周期内，某entity处于runnable状态的时间为x，那么其对系统负载的贡献为（x/1024）；pelt还会累积过去周期的负载贡献，但会根据时间乘以相应的衰减系数y，假设Li表示某entity在周期Pi中对系统负载的贡献，那么该entity对系统负载的总贡献如下：

每经过32个周期，entity对系统负载的贡献减半

1.2 重要数据结构

内核使用struct sched\_avg结构体记录se和cfs\_rq的负载贡献情况：

struct sched\_entity {

struct sched\_avg avg;

};

struct cfs\_rq {

struct sched\_avg avg;

};

结构体定义如下：

struct sched\_avg {

/\* 上一次更新的时间 \*/

u64 last\_update\_time;

u64 load\_sum;

u64 runnable\_sum;

u32 util\_sum;

/\* 上次更新时，未满一个pelt计算周期（1024us）的部分 \*/

u32 period\_contrib;

unsigned long load\_avg;

unsigned long runnable\_avg;

unsigned long util\_avg;

/\* 用于估算CFS进程的利用率，本文暂不讨论，感兴趣的可以参考

\* commit 7f65ea42eb00b(sched/fair: Add util\_est on top of PELT)

\*/

struct util\_est util\_est;

} \_\_\_\_cacheline\_aligned;

有三组统计负载贡献的成员：

load\_{sum|avg}：记录runnable状态的负载贡献情况

runnable\_{sum|avg}：为load\_{sum|avg}的去weight版本，该成员在今年二月份加入内核（sched/pelt: Add a new runnable average signal ）

util\_{sum|avg}：记录running状态下的负载贡献情况

static \_\_always\_inline u32

accumulate\_sum(u64 delta, struct sched\_avg \*sa,

unsigned long load, unsigned long runnable, int running)

{

u32 contrib = (u32)delta; /\* p == 0 -> delta < 1024 \*/

u64 periods;

delta += sa->period\_contrib;

periods = delta / 1024; /\* A period is 1024us (~1ms) \*/

/\*

\* Step 1: decay old \*\_sum if we crossed period boundaries.

\*/

if (periods) {

sa->load\_sum = decay\_load(sa->load\_sum, periods);

sa->runnable\_sum =

decay\_load(sa->runnable\_sum, periods);

sa->util\_sum = decay\_load((u64)(sa->util\_sum), periods);

/\*

\* Step 2

\*/

delta %= 1024;

if (load) {

/\*

\* This relies on the:

\*

\* if (!load)

\* runnable = running = 0;

\*

\* clause from \_\_\_update\_load\_sum(); this results in

\* the below usage of @contrib to disappear entirely,

\* so no point in calculating it.

\*/

contrib = \_\_accumulate\_pelt\_segments(periods,

1024 - sa->period\_contrib, delta);

}

}

sa->period\_contrib = delta;

**if (load)**

**sa->load\_sum += load \* contrib;**

**if (runnable)**

**sa->runnable\_sum += runnable \* contrib << SCHED\_CAPACITY\_SHIFT;**

**if (running)**

**sa->util\_sum += contrib << SCHED\_CAPACITY\_SHIFT;**

return periods;

}

#### UTIL\_EST

由于在 PELT 算法下任务的 util 增加减少的都比较慢，对于长时间休眠后的重负载任务，其 util 增加的比较慢，导致不能及时触发提频和迁核。为了补救 PELT 的这一缺陷，引入了预估负载。在任务(休眠)出队列时更新任务的预估负载，当任务入队列时将出队列时的负载加到cfs\_rq的预估负载上

### WALT

[调度器6—WALT负载计算 - Hello-World3 - 博客园 (cnblogs.com)](https://www.cnblogs.com/hellokitty2/p/15376806.html)

## 负载均衡

## EAS

## IPA

# 内核同步对象

## mutex->futex

// 声明一个互斥量

pthread\_mutex\_t mtx;

// 初始化

pthread\_mutex\_init(&mtx, NULL);

// 加锁

pthread\_mutex\_lock(&mtx);

// 解锁

pthread\_mutex\_unlock(&mtx);

// 销毁

pthread\_mutex\_destroy(&mtx);

## condition

// 声明一个互斥量

pthread\_mutex\_t mtx;

// 声明一个条件变量

pthread\_cond\_t cond;

...

// 初始化

pthread\_mutex\_init(&mtx, NULL);

pthread\_cond\_init(&cond, NULL);

// 加锁

pthread\_mutex\_lock(&mtx);

// 加锁成功，等待条件变量触发

pthread\_cond\_wait(&cond, &mtx);

...

// 加锁

pthread\_mutex\_lock(&mtx);

pthread\_cond\_signal(&cond);

...

// 解锁

pthread\_mutex\_unlock(&mtx);

// 销毁

pthread\_mutex\_destroy(&mtx);

## rwlock

// 声明一个读写锁

pthread\_rwlock\_t rwlock;

...

// 在读之前加读锁

pthread\_rwlock\_rdlock(&rwlock);

... 共享资源的读操作

// 读完释放锁

pthread\_rwlock\_unlock(&rwlock);

// 在写之前加写锁

pthread\_rwlock\_wrlock(&rwlock);

... 共享资源的写操作

// 写完释放锁

pthread\_rwlock\_unlock(&rwlock);

// 销毁读写锁

pthread\_rwlock\_destroy(&rwlock);

## semaphore

## spinlock

[(17条消息) Linux 内核同步（二）：自旋锁（Spinlock）\_spin\_lock\_爱洋葱的博客-CSDN博客](https://blog.csdn.net/zhoutaopower/article/details/86598839)

spin\_lock 的时候，禁止内核抢占

如果涉及到中断上下文的访问，spin lock需要和禁止本 CPU 上的中断联合使用（spin\_lock\_irqsave / spin\_unlock\_irqstore）

涉及 half bottom 使用：spin\_lock\_bh / spin\_unlock\_bh

内核当发生访问资源冲突的时候，可以有两种锁的解决方案选择：

一个是原地等待

一个是挂起当前进程，调度其他进程执行（睡眠）

Spinlock 是内核中提供的一种比较常见的锁机制，自旋锁是“原地等待”的方式解决资源冲突的，即，一个线程获取了一个自旋锁后，另外一个线程期望获取该自旋锁，获取不到，只能够原地“打转”（忙等待）。由于自旋锁的这个忙等待的特性，注定了它使用场景上的限制 —— 自旋锁不应该被长时间的持有（消耗 CPU 资源）

在linux kernel的实现中，经常会遇到这样的场景：共享数据被中断上下文和进程上下文访问，该如何保护呢？如果只有进程上下文的访问，那么可以考虑使用semaphore或者mutex的锁机制，但是现在中断上下文也参和进来，那些可以导致睡眠的lock就不能使用了，这时候，可以考虑使用spin lock

自旋锁的使用

在linux kernel的实现中，经常会遇到这样的场景：共享数据被中断上下文和进程上下文访问，该如何保护呢？如果只有进程上下文的访问，那么可以考虑使用semaphore或者mutex的锁机制，但是现在中断上下文也参和进来，那些可以导致睡眠的lock就不能使用了，这时候，可以考虑使用spin lock。

这里为什么把中断上下文标红加粗呢？因为在中断上下文，是不允许睡眠的（原因详见文章《Linux 中断之中断处理浅析》中的第四章），所以，这里需要的是一个不会导致睡眠的锁——spinlock。

换言之，中断上下文要用锁，首选 spinlock。

使用自旋锁，有两种方式定义一个锁：

动态的：

spinlock\_t lock;

spin\_lock\_init (&lock);

静态的：

DEFINE\_SPINLOCK(lock);

自旋锁的死锁和解决

自旋锁不可递归，自己等待自己已经获取的锁，会导致死锁。

自旋锁可以在中断上下文中使用，但是试想一个场景：一个线程获取了一个锁，但是被中断处理程序打断，中断处理程序也获取了这个锁（但是之前已经被锁住了，无法获取到，只能自旋），中断无法退出，导致线程中后面释放锁的代码无法被执行，导致死锁。（如果确认中断中不会访问和线程中同一个锁，其实无所谓）

一、考虑下面的场景（内核抢占场景）：

（1）进程A在某个系统调用过程中访问了共享资源 R

（2）进程B在某个系统调用过程中也访问了共享资源 R

会不会造成冲突呢？假设在A访问共享资源R的过程中发生了中断，中断唤醒了沉睡中的，优先级更高的B，在中断返回现场的时候，发生进程切换，B启动执行，并通过系统调用访问了R，如果没有锁保护，则会出现两个thread进入临界区，导致程序执行不正确。OK，我们加上spin lock看看如何：A在进入临界区之前获取了spin lock，同样的，在A访问共享资源R的过程中发生了中断，中断唤醒了沉睡中的，优先级更高的B，B在访问临界区之前仍然会试图获取spin lock，这时候由于A进程持有spin lock而导致B进程进入了永久的spin……怎么破？linux的kernel很简单，在A进程获取spin lock的时候，禁止本CPU上的抢占（上面的永久spin的场合仅仅在本CPU的进程抢占本CPU的当前进程这样的场景中发生）。如果A和B运行在不同的CPU上，那么情况会简单一些：A进程虽然持有spin lock而导致B进程进入spin状态，不过由于运行在不同的CPU上，A进程会持续执行并会很快释放spin lock，解除B进程的spin状态

二、再考虑下面的场景（中断上下文场景）：

（1）运行在CPU0上的进程A在某个系统调用过程中访问了共享资源 R

（2）运行在CPU1上的进程B在某个系统调用过程中也访问了共享资源 R

（3）外设P的中断handler中也会访问共享资源 R

在这样的场景下，使用spin lock可以保护访问共享资源R的临界区吗？我们假设CPU0上的进程A持有spin lock进入临界区，这时候，外设P发生了中断事件，并且调度到了CPU1上执行，看起来没有什么问题，执行在CPU1上的handler会稍微等待一会CPU0上的进程A，等它立刻临界区就会释放spin lock的，但是，如果外设P的中断事件被调度到了CPU0上执行会怎么样？CPU0上的进程A在持有spin lock的状态下被中断上下文抢占，而抢占它的CPU0上的handler在进入临界区之前仍然会试图获取spin lock，悲剧发生了，CPU0上的P外设的中断handler永远的进入spin状态，这时候，CPU1上的进程B也不可避免在试图持有spin lock的时候失败而导致进入spin状态。为了解决这样的问题，linux kernel采用了这样的办法：如果涉及到中断上下文的访问，spin lock需要和禁止本 CPU 上的中断联合使用。

三、再考虑下面的场景（底半部场景）

linux kernel中提供了丰富的bottom half的机制，虽然同属中断上下文，不过还是稍有不同。我们可以把上面的场景简单修改一下：外设P不是中断handler中访问共享资源R，而是在的bottom half中访问。使用spin lock+禁止本地中断当然是可以达到保护共享资源的效果，但是使用牛刀来杀鸡似乎有点小题大做，这时候disable bottom half就OK了

四、中断上下文之间的竞争

同一种中断handler之间在uni core和multi core上都不会并行执行，这是linux kernel的特性。

如果不同中断handler需要使用spin lock保护共享资源，对于新的内核（不区分fast handler和slow handler），所有handler都是关闭中断的，因此使用spin lock不需要关闭中断的配合。

bottom half又分成softirq和tasklet，同一种softirq会在不同的CPU上并发执行，因此如果某个驱动中的softirq的handler中会访问某个全局变量，对该全局变量是需要使用spin lock保护的，不用配合disable CPU中断或者bottom half

tasklet更简单，因为同一种tasklet不会多个CPU上并发

## RCU

RCU(Read-Copy Update)，顾名思义就是读-拷贝修改，它是基于其原理命名的。对于被RCU保护的共享数据结构，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，最后使用一个回调（callback）机制在适当的时机把指向原来数据的指针替换为新的被修改的数据。这个时机就是所有引用该数据的CPU都退出对共享数据的访问

在多线程场景下，经常我们需要并发访问一个数据结构，为了保证线程安全我们会考虑使用互斥设施来进行同步，更进一步我们会根据对这个数据结构的读写比例而选用读写锁进行优化。但是读写锁不是唯一的方式，我们可以借助于COW技术来做到写操作不需要加锁，也就是在读的时候正常读，写的时候，先加锁拷贝一份，然后进行写，写完就原子的更新回去，使用COW实现避免了频繁加读写锁本身的性能开销

RCU锁的核心思想：

读者无锁访问数据，标记进出临界区；

写者读取，复制，更新；

旧数据延迟回收

# 中断

## 中断定义

中断是指在CPU正常运行期间，由于内外部事件或由程序预先安排的事件引起的 CPU 暂时停止正在运行的程序，转而为该内部或外部事件或预先安排的事件服务的程序中去，服务完毕后再返回去继续运行被暂时中断的程序。Linux中通常分为外部中断（又叫硬件中断）和内部中断（又叫异常）。

软件对硬件进行配置后，软件期望等待硬件的某种状态（比如，收到了数据），这里有两种方式，一种是轮询（polling）： CPU 不断的去读硬件状态。另一种是当硬件完成某种事件后，给 CPU 一个中断，让 CPU 停下手上的事情，去处理这个中断。很显然，中断的交互方式提高了系统的吞吐。

当 CPU 收到一个中断 （IRQ）的时候，会去执行该中断对应的处理函数（ISR）。普通情况下，会有一个中断向量表，向量表中定义了 CPU 对应的每一个外设资源的中断处理程序的入口，当发生对应的中断的时候， CPU 直接跳转到这个入口执行程序。也就是中断上下文。（注意：中断上下文中，不可阻塞睡眠）

## 中断TOP&BOTTOM

中断服务程序的设计最好是快速完成任务并退出，因为此刻系统处于被中断中。但是在 ISR 中又有一些必须完成的事情，比如：清中断标志，读/写数据，寄存器操作等。

在 Linux 中，同样也是这个要求，希望尽快的完成 ISR。但事与愿违，有些 ISR 中任务繁重，会消耗很多时间，导致响应速度变差。Linux 中针对这种情况，将中断分为了两部分：

1. 上半部（top half）：收到一个中断，立即执行，有严格的时间限制，只做一些必要的工作，比如：应答，复位等。这些工作都是在所有中断被禁止的情况下完成的

2. 底半部（bottom half）：能够被推迟到后面完成的任务会在底半部进行。在适合的时机，下半部会被开中断执行。（具体的机制在下一篇文章分析）

## 中断处理程序

驱动程序可以使用接口：

request\_irq(unsigned int irq, irq\_handler\_t handler, unsigned long flags, const char \*name, void \*dev)

像系统申请注册一个中断处理程序。

其中的参数：

参数 含义

irq 表了该中断的中断号，一般 CPU 的中断号都会事先定义好。

handler 中断发生后的 ISR

flags 中断标志( IRQF\_DISABLED / IRQFSAMPLE\_RANDOM / IRQF\_TIMER / IRQF\_SHARED)

name 中断相关的设备 ASCII 文本，例如 "keyboard"，这些名字会在 /proc/irq 和 /proc/interrupts 文件使用

dev 用于共享中断线，传递驱动程序的设备结构。非共享类型的中断，直接设置成为 NULL

中断标志 flag 的含义：

标志 含义

IRQF\_DISABLED 设置这个标志的话，意味着内核在处理这个 ISR 期间，要禁止其他中断（多数情况不使用这个）

IRQFSAMPLE\_RANDOM 表明这个设备产生的中断对内核熵池有贡献

IRQF\_TIMER 为系统定时器准备的标志

IRQF\_SHARED 表明多个中断处理程序之间共享中断线。同一个给定的线上注册每个处理程序，必须设置这个

调用 request\_irq 成功执行返回 0。常见错误是 -EBUSY，表示给定的中断线已经在使用（或者没有指定 IRQF\_SHARED）

注意：request\_irq 函数可能引起睡眠，所以不允许在中断上下文或者不允许睡眠的代码中调用

释放中断：

const void \*free\_irq(unsigned int irq, void \*dev\_id)

用于释放中断处理函数

注意：Linux 中的中断处理程序是无须重入的。当给定的中断处理程序正在执行的时候，其中断线在所有的处理器上都会被屏蔽掉，以防在同一个中断线上又接收到另一个新的中断。通常情况下，除了该中断的其他中断都是打开的，也就是说其他的中断线上的重点都能够被处理，但是当前的中断线总是被禁止的，故，同一个中断处理程序是绝对不会被自己嵌套的

## 中断上下文

与进程上下文不一样，内核执行中断服务程序的时候，处于中断上下文。中断处理程序并没有自己的独立的栈，而是使用了内核栈，其大小一般是有限制的（32bit 机器 8KB）。所以其必须短小精悍。同时中断服务程序是打断了正常的程序流程，这一点上也必须保证快速的执行。同时中断上下文中是不允许睡眠，阻塞的

中断上下文不能睡眠的原因是：

1、 中断处理的时候,不应该发生进程切换，因为在中断context中，唯一能打断当前中断handler的只有更高优先级的中断，它不会被进程打断，如果在 中断context中休眠，则没有办法唤醒它，因为所有的wake\_up\_xxx都是针对某个进程而言的，而在中断context中，没有进程的概念，没有一个task\_struct（这点对于softirq和tasklet一样），因此真的休眠了，比如调用了会导致block的例程，内核几乎肯定会死

2、schedule()在切换进程时，保存当前的进程上下文（CPU寄存器的值、进程的状态以及堆栈中的内容），以便以后恢复此进程运行。中断发生后，内核会先保存当前被中断的进程上下文（在调用中断处理程序后恢复）；

但在中断处理程序里，CPU寄存器的值肯定已经变化了吧（最重要的程序计数器PC、堆栈SP等），如果此时因为睡眠或阻塞操作调用了schedule()，则保存的进程上下文就不是当前的进程context了.所以不可以在中断处理程序中调用schedule()

3、内核中schedule()函数本身在进来的时候判断是否处于中断上下文:

if(unlikely(in\_interrupt()))

BUG();

因此，强行调用schedule()的结果就是内核BUG

4、中断handler会使用被中断的进程内核堆栈，但不会对它有任何影响，因为handler使用完后会完全清除它使用的那部分堆栈，恢复被中断前的原貌

5、处于中断context时候，内核是不可抢占的。因此，如果休眠，则内核一定挂起

# 内存管理

## COW（COPY ON WRITE）

缺页中断是内核在访问一个虚拟地址时，发现该虚拟地址所在的页没有一个物理页框与之相对应，就会触发缺页异常进入物理页面分配的异常函数。而写时复制是指，父进程在fork出子进程的时候，并不会为子进程马上分配物理页框，而是当复进程或者子进程需要对内存进行写操作时，才会分配物理页框，分配时，先将内容完全复制到新页框，然后再进行改写，称为写时复制

# CPU架构

## 微架构

### 前端

#### L1&L2的带宽和Latency

#### TLB的带宽和Latency

#### 译码宽度

#### MOP Cache

对于Dhrystone这种指令规模比较小的来说，MOP可以显著加速译码速度，但是对于应用场景，很难命中，放在上面增加功耗和面积

MOP Cache的边界效应比较明显

#### 分支预测

分支预测器的精度要和预取宽度相适应，否则预测失败导致的流水线清空惩罚会比较大

### 后端

#### ROB

#### ALU

整数、浮点、向量SVE2/SVE/NEON

#### Issue Width

#### Rename

#### LSQ

## 拓扑结构

44/134/224/1322/244/11224/143/152

### 多丛集群调度

Z3 采用五丛集群调度：

[E1@3.2GHz](mailto:E1@3.2GHz) HunterELP高频能效冲跑分

[E0@2.45GHz](mailto:E0@2.45GHz) HunterELP 低频能效，应用启动安装等单线程、重载游戏

[B23@2.55GHz](mailto:B23@2.55GHz) Hunter高频能效 重载游戏

[B01@1.8GHz](mailto:B01@1.8GHz) Hunter低频能效 王者荣耀

[L0123@1.6](mailto:L0123@1.6) Hayes 多后台、低负载

Z4：

224 2Blackhawk+2Chaberton+4ChabtonMini

## 供电

供电设计需要考虑共PowerRail问题

### 小核和DSU

### ELP和大核

## Cache

Cache设计考虑如何提高Cache命中率，实际上是用来解决程序工作集较大或者局部性很差的问题，典型的用户场景如应用启动/安装/滑动/游戏：预取和替换算法、边界效应（随Size增加命中率提升不明显）

硬件设计上考虑：增加硬件预取器比如L3Prefetcher，在specint2006中访存类型的用例很多，L3Prefetcher在论文中预计有3%的优化幅度，但是对于GeekBench5，L3影响很小，增加L3不会带来较大的性能收益

### 硬件设计

Way/Set/CacheLine

### 供电设计

如果Cache比较大的话，需要考虑分区下电，典型的如L3

### Latency

不同用户模式的访存延迟，是评价Cache的关键性能指标

### 预取算法

### 替换算法

LRU

### TLB

TLB存放页表基地址，MMU通过查询TLB完成虚拟地址到物理地址的转换，TLB Miss会导致两次物理内存访问

其中L1 iTLB和dTLB是分开的，L2的是混合的

### L2

### L3

### SLC

## 拓扑结构

## 制程

N3E技术是目前第一代3纳米技术（N3）的升级版，预计将于今年开始进行测试，明年下半年开始批量生产，苹果目前正在研发的A17和M3芯片将使用这一工艺制程。

据说对比台积电5纳米技术（N5），N3E在同等性能和密度下功耗降低34％、同等功耗和密度下性能提升18％，可以将晶体管密度提升60％。

其实台积电N4工艺的升级就比较大，采用这一制程工艺的A16处理器，晶体管从150亿增加到160亿，性能也有了很大升级，可以期待一下A17的性能表现

## ELP（Enhanced Lead Partner）

## FloorPlan

Floorplan，中文翻译是布局规划。位于后端设计的最前端

## ARM RoadMap

### TCS23

<https://www.anandtech.com/show/18871/arm-unveils-armv92-mobile-architecture-cortex-x4-a720-and-a520-64bit-exclusive/6>

8650:1HunterELP+5Hunter+2Hayes

DX3/9300:4HunterELP+4Hunter

台积电N3E工艺（宣称主攻能效，但实际是没达到预期目标的3nm工艺，但肯定比N4/N4P强）不再兼容32位APP，宣称全系统14%能效提升，DSU功耗降18%，最多14核，10个X4+4个A720，最高32M的L3缓存

X4同性能功耗降40%，IPC同频性能提15%，面积增加10%。L2最大2MB

A720同性能功耗降20%，同频性能提升15%

A520同性能功耗降22%，同频性能提升8%

G720峰值性能和持续性能提升15%，降低降低40%带宽占用

### TCS24

8750/自研

DX4/9400

X5(Blackhawk) L2最大4MB

A730(Chaberton)

A520(Hayes)

G730(Krake)

### TCS25

8850/自研

DX5/9500

X6()

A740()

A540()

G740()

# GPU架构

# 性能建模

## GEM5模型校准

1. 算力校准Dhrystone
2. Latency校准LMBench
3. 分支预测算法
4. MOP Cache实现
5. L1/L2/L3 Cache预取替换算法、Inclusive/Exclusive
6. TLB预取替换算法
7. CPU/SLC/DDR BUS Hash
8. SPEC切片校准
9. Kernel5.15+AndroidT BringUp
10. 单机Android应用指令切片，安装/启动/游戏/滑动
11. 支持Network，Android应用联网运行
12. 支持TP操作Android界面
13. DynamicRIO在9200/8550进行指令切片，如果10-12不能实现的话
14. 真机指令切片GEM5回放
15. ARMV9支持

## Android应用程序能耗分析与建模研究

应用程序能耗分析与建模是智能移动终端能耗优化的重要组成部分。针对智能移动终端丰富的应用程序,提出了一种基于应用程序运行时间的时间能耗模型。与精度高和复杂的应用程序组件能耗模型相比,该模型使用时间变量刻画和包含终端的功耗、性能等多种因素,而且运行时间容易精确测量和获取,能够快速地估算应用程序运行时移动终端产生的能耗

<https://wenku.baidu.com/view/f7f89d994028915f814dc218?fr=xueshu&_wkts_=1686965937030>

## EAS

择一个任务Util放置在哪个核上，是由能效曲线的斜率决定的，即最佳能效比

ARM提供的能效曲线，是Power和Capacity，EAS选核的时候是对比能量E最低而不是功率P最低，需要计算运行时间

energy = power \* time

energy = power \* util/cap\_cur\_freq //选择一个任务Util放置在哪个核上，是由能效曲线的斜率决定的

energy = power \* util/(freq/freq\_max \* scale\_cpu)

energy = power \* util \* freq\_max/(freq \* scale\_cpu)

energy = cost \* util / scale\_cpu

energy = power/scale\_cpu \* freq\_max/freq \* util

energy = power/(freq/freq\_max\*scale\_cpu) \* util

energy = C\*V\*V\*freq/(freq/freq\_max\*scale\_cpu) \* util

energy = C\*V\*V\*freq\_max/scale\_cpu \* util

考虑静态功耗的情况下：

energy = (Pdynamic+Pstatic)/(freq/freq\_max\*scale\_cpu) \* util

energy = C\*V\*V\*freq\_max/scale\_cpu \* util +Pstatic/(freq/freq\_max\*scale\_cpu) \* util

energy = C\*V\*V\*freq\_max/scale\_cpu \* util +P(V、Thermal)/(freq/freq\_max\*scale\_cpu) \* util

energy = C\*V\*V\*freq\_max/scale\_cpu \* util +P(V、Thermal)\*freq\_max/(freq\*scale\_cpu) \* util

### 选核硬化

### EAS选核流程硬化

## DPM

通过DPM的PMU Events建模得到CPU、GPU功耗和温度

## AIPNP

通过simpleperf直接抓取王者荣耀的PMU Events，通过AI Training得到量化模型，运行时无需采样，得到目标频率

## IPA

## CLPC：PID

## 负载跟踪和选核硬化

## Ux跟踪任务唤醒依赖关系硬化

## 16K大页

## CPU Latency随带宽增加而增加，游戏场景的延迟是否更大，增强CPU的OutStanding/Burst能力

## CPU/GPU/SLC/DDR Hash

## QoS

降低CPU的访存延迟

## MPAM L3/SLC

## PDM&MPMM

## CPU BYPASS SLC

## ZRAM硬化

## L3 HWPrefetcher

## AI HWPrefetcher

<https://kns.cnki.net/kcms2/article/abstract?v=3uoqIhG8C475KOm_zrgu4lQARvep2SAkueNJRSNVX-zc5TVHKmDNkpZmHIXVmQyUpJe4efasu7XRtHFMOtg8Khq6xvbyqoCe&uniplatform=NZKPT>

## 能效

### CPU硬件拓扑结构

如何确定下列组合能效最优：

134、143、152、1322、224、242、44、11224、244、

设计基本原则是能效曲线必须连续，中间不能断开

考虑AVS

考虑静态功耗

考虑EAS选核E=C\*V\*V\*F\*T + L(V&T)，运行时间T是占空比，不同IDLE等级下功耗的不计算

Dhrystone功耗、SPEC算力、SPEC功耗

各Cluster的FMax/电压/频率/功耗(动态+静态)/共电源域/温度(65/85)

场景为Benchmark和DoU

Benchmark：GeekBench、Antutu

DoU场景的功耗：

1. 屏幕：刷新率60/90/120，分辨率2K/3K/4K
2. CPU：滑动、视频播放、直播、启动
3. Modem/WCN：4G/5G/WIFI下载
4. DDR
5. DSU
6. SLC
7. GPU
8. Sensor
9. ISP