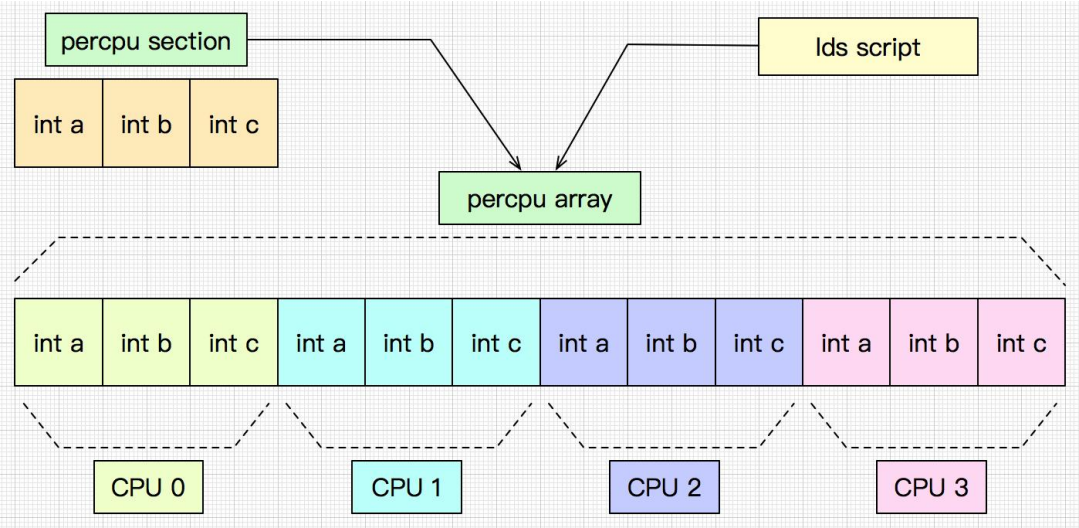
作者：代兴　邮箱：503268771@qq.com　Vx 公众号：东方架构师
https://github.com/drink-cat/Book_Program-Principles

percpu 的含义和示意图

percpu 是 `per cpu param` 的缩写，表示每个 CPU 有独立的变量空间，实现 CPU 隔离。（这里的 CPU 指 CPU 核）
percpu 是提高并发、优化性能的重要方式。linux 把任务调度到 CPU 执行，每个 CPU 操作自己的 percpu 变量。因为 percpu 变量不会被多个 CPU 共享，所以不需要互斥锁。
linux 源码使用了很多 percpu，实现极致性能优化。
上层软件也可以使用自定义 percpu，关键在于找到 CPU 编号，分配 percpu 变量。
percpu 分为静态 percpu、动态 percpu 等。这里分析静态 percpu。

percpu 的示意图：



percpu 的实现过程：
使用 LDS 脚本，收集 percpu 变量，放入 percpu section。
使用 CPU 数量和 percpu section 大小，生成 percpu 数组。
使用当前的 CPU 编号和 percpu 变量的地址偏移，获得 percpu 变量的指针，就能读写 percpu 变量。

linux 源码中的 percpu

文件 linux-5.6.3/include/asm-generic/percpu.h。

```
#define PER_CPU_BASE_SECTION ".data..percpu"

/*
 * per_cpu_offset() is the offset that has to be added to a
 * percpu variable to get to the instance for a certain processor.
 *
 * Most arches use the __per_cpu_offset array for those offsets but
 * some arches have their own ways of determining the offset (x86_64, s390).
 */
```

```
#ifndef __per_cpu_offset
extern unsigned long __per_cpu_offset[NR_CPUS];

#define per_cpu_offset(x) (__per_cpu_offset[x])
#endif

#define __my_cpu_offset per_cpu_offset(raw_smp_processor_id())

#define arch_raw_cpu_ptr(ptr) SHIFT_PERCPU_PTR(ptr, __my_cpu_offset)
```

文件 linux-5.6.3/include/linux/percpu-defs.h。

```
#define DEFINE_PER_CPU(type, name)                  \
    DEFINE_PER_CPU_SECTION(type, name, "")

#define DEFINE_PER_CPU_SECTION(type, name, sec)             \
    __PCPU_ATTRS(sec) __typeof__(type) name

#define __PCPU_ATTRS(sec)                       \
    __percpu __attribute__((section(PER_CPU_BASE_SECTION sec))) \
    PER_CPU_ATTRIBUTES

#define SHIFT_PERCPU_PTR(__p, __offset)                 \
    RELOC_HIDE((typeof(*(__p)) __kernel __force *)(__p), (__offset))

#define per_cpu_ptr(ptr, cpu)                       \
({                                  \
    __verify_pcpu_ptr(ptr);                     \
    SHIFT_PERCPU_PTR((ptr), per_cpu_offset((cpu)));         \
})

#define per_cpu(var, cpu)   (*per_cpu_ptr(&(var), cpu))
```

文件 linux-5.6.3/drivers/clocksource/timer-riscv.c。

```
static DEFINE_PER_CPU(struct clock_event_device, riscv_clock_event) = {
    .name           = "riscv_timer_clockevent",
    .features       = CLOCK_EVT_FEAT_ONESHOT,
    .rating         = 100,
    .set_next_event     = riscv_clock_next_event,
};

/* called directly from the low-level interrupt handler */
void riscv_timer_interrupt(void)
{
    struct clock_event_device *evdev = this_cpu_ptr(&riscv_clock_event);

    csr_clear(CSR_IE, IE_TIE);
    evdev->event_handler(evdev);
}
```

文件 linux-5.6.3/arch/ia64/kernel/irq_ia64.c。

```
DEFINE_PER_CPU(int[IA64_NUM_VECTORS], vector_irq) = {
    [0 ... IA64_NUM_VECTORS - 1] = -1
};

void __setup_vector_irq(int cpu)
{
    int irq, vector;

    /* Clear vector_irq */
    for (vector = 0; vector < IA64_NUM_VECTORS; ++vector)
        per_cpu(vector_irq, cpu)[vector] = -1;
    /* Mark the inuse vectors */
    for (irq = 0; irq < NR_IRQS; ++irq) {
        if (!cpumask_test_cpu(cpu, &irq_cfg[irq].domain))
            continue;
        vector = irq_to_vector(irq);
        per_cpu(vector_irq, cpu)[vector] = irq;
    }
}
```

使用 C 和汇编实现自定义的 percpu

编写代码： percpu.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <stdint.h>
#include <string.h>
#include <pthread.h>

// cpu 核的总数
#define cpu_total_count() sysconf(_SC_NPROCESSORS_ONLN)
// 当前的 cpu 编号
#define cpu_current_no() sched_getcpu()

struct cat_t
{
    uint32_t color;
    uint32_t speed;
};

// 自定义 section
__attribute__((section("my_percpu"))) struct cat_t per_cat;
__attribute__((section("my_percpu"))) uint64_t per_load;
```

```
uint32_t per_begin; // 记录 section 的开始地址
uint32_t per_end;   // 记录 section 的结束地址

int cpu_total;    // cpu 核的总数
int section_size; // 自定义 section 的大小
void *array_ptr;  // 数组

// 生成数组
void build_array()
{
    // 自定义 section 的大小
    section_size = (uint64_t)&per_end - (uint64_t)&per_begin;
    // 数组大小
    int array_size = section_size * cpu_total;
    // 申请整块内存
    array_ptr = malloc(array_size);
    memset(array_ptr, 0, array_size);
    printf(" cpu_total    = %d \n", cpu_total);
    printf(" section_size = %d \n", section_size);
    printf(" array_ptr    = %p \n\n", array_ptr);
}

// 获得 percpu 的变量
struct cat_t *get_per_cat()
{
    // 获得当前的 CPU 编号
    int cpu_curr = cpu_current_no();
    // 找到 CPU 编号对应的 percpu 数组元素
    void *array_element = array_ptr + section_size * cpu_curr;
    // 找到变量的地址偏移
    int offset = (uint64_t)&per_cat - (uint64_t)&per_begin;
    // 找到变量的地址
    void *cat_addr = array_element + offset;
    // 获得变量指针
    return (struct cat_t *)cat_addr;
}

// 获得 percpu 的变量
uint64_t *get_per_load()
{
    // 当前 CPU 编号
    int cpu_curr = cpu_current_no();
    // 找到 CPU 编号对应的数组元素
    void *array_element = array_ptr + section_size * cpu_curr;
    // 找到地址偏移
    int offset = (uint64_t)&per_load - (uint64_t)&per_begin;
    // 找到变量地址
    void *count_addr = array_element + offset;
    // 转换类型
```

```
    return (uint64_t *)count_addr;
}

// 打印 percpu 的数组
void print_array()
{
    printf("\n");
    // 遍历每个 CPU
    for (int n = 0; n < cpu_total; ++n)
    {
        // 数组元素
        void *array_element = array_ptr + section_size * n;
        // 找到地址偏移
        int offset_cat = (uint64_t)&per_cat - (uint64_t)&per_begin;
        int offset_load = (uint64_t)&per_load - (uint64_t)&per_begin;
        // 找到变量地址
        struct cat_t *cat = (struct cat_t *)(array_element + offset_cat);
        uint64_t *load = (uint64_t *)(array_element + offset_load);
        printf(" CPU = %d  array_element = %p  load = %llu  cat.speed = %u \n",
               n, array_element, *load, cat->speed);
    }
}

// 线程的函数
void *thread_func(void *param)
{
    char *name = (char *)param;
    for (int m = 0; m < 3; ++m)
    {
        // 当前 CPU 编号
        int cpu_curr = cpu_current_no();
        printf(" %s on CPU = %d .", name, cpu_curr);

        // 增加负载
        uint64_t *load = get_per_load();
        printf(" load = %p .", load);
        ++(*load);

        // 和上一步相同操作，方便对比。
        struct cat_t *cat = get_per_cat();
        printf(" cat = %p \n", cat);
        ++(cat->speed);

        // 手动触发线程调度。可能调度到不同的 CPU
        sleep(1);
    }
}

int main()
```

```
{
    cpu_total = cpu_total_count();
    // 构建数组
    build_array();

    pthread_t t1;
    pthread_create(&t1, NULL, thread_func, "Thread-1");
    pthread_t t2;
    pthread_create(&t2, NULL, thread_func, "Thread-2");

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    print_array();
    free(array_ptr);
    return 0;
}
```

编写脚本： percpu.lds

```
SECTIONS {
    section_my_percpu :  /* 自定义 section */
    {
        per_begin = . ;  /* 开始地址 */
        *(my_percpu) ;   /* 收集符号 */
        per_end = . ;    /* 结束地址 */
    }
}
```

编译代码：

```
gcc percpu.c percpu.lds -lpthread -std=gnu99 -o percpu
readelf -a percpu > percpu.elf.txt
```

运行代码：

```
[root@local percpu]# ./percpu
 cpu_total    = 3
 section_size = 16
 array_ptr    = 0x13a5010

 Thread-1 on CPU = 0 . load = 0x13a5018 . cat = 0x13a5010
 Thread-2 on CPU = 2 . load = 0x13a5038 . cat = 0x13a5030
 Thread-2 on CPU = 1 . load = 0x13a5028 . cat = 0x13a5020
 Thread-1 on CPU = 1 . load = 0x13a5028 . cat = 0x13a5020
 Thread-2 on CPU = 1 . load = 0x13a5028 . cat = 0x13a5020
 Thread-1 on CPU = 2 . load = 0x13a5038 . cat = 0x13a5030

 CPU = 0  array_element = 0x13a5010  load = 1  cat.speed = 1
 CPU = 1  array_element = 0x13a5020  load = 3  cat.speed = 3
 CPU = 2  array_element = 0x13a5030  load = 2  cat.speed = 2
```

分析结果：
查看文件 percpu.elf.txt，找到 percpu 的 section 和 symbol。
section_my_percpu 的编号是 27。变量 per_begin、per_cat、per_end、per_load 所在的 section 的编号是 27。section 正常匹配。

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [27] section_my_percpu PROGBITS         0000000000a020a0  000020a0
       0000000000000010  0000000000000000  WA       0     0     8

Symbol table '.symtab' contains 85 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
    58: 0000000000a020a0     4 OBJECT  GLOBAL DEFAULT   27 per_begin
    62: 0000000000a020a0     8 OBJECT  GLOBAL DEFAULT   27 per_cat
    75: 0000000000a020b0     4 OBJECT  GLOBAL DEFAULT   27 per_end
    82: 0000000000a020a8     8 OBJECT  GLOBAL DEFAULT   27 per_load
```

查看运行结果。
CPU 数量为 3，percpu section 的大小为 16 字节，percpu 数组的地址为 0x13a5010。
线程 1 在 CPU 0、CPU 1、CPU 2 上面运行。线程 2 在 CPU 1、CPU 2 上面运行。
CPU 运行次数不均匀。CPU 0 运行了 1 次，CPU 1 运行了 3 次，CPU 2 运行了 2 次。
每个 CPU 的 percpu 变量，load 等于 cat.speed，读写过程没有加锁，没有发生覆盖写。

percpu 的实现步骤：
第一步，生成 percpu section。
使用 section 标记修饰 percpu 变量，`__attribute__((section("my_percpu"))) struct cat_t per_cat;`。
使用 lds 脚本 `*(my_percpu)`，把 percpu 变量放入一个 section。
使用变量 per_begin 记录 section 的开始地址，`per_begin = . ;`。
使用变量 per_end 记录 section 的结束地址，`per_end = . ;`。
第二步，生成 percpu 数组。
计算 section 占用的大小，`section_size = (uint64_t)&per_end - (uint64_t)&per_begin`。
获得 CPU 数量，`#define cpu_total_count() sysconf(_SC_NPROCESSORS_ONLN)`。
计算 percpu 数组的大小，`int array_size = section_size * cpu_total;`。
分配 percpu 数组，`array_ptr = malloc(array_size);`。
清零 percpu 数组，`memset(array_ptr, 0, array_size);`。
第三步，获得 percpu 变量。
获得当前的 CPU 编号，`#define cpu_current_no() sched_getcpu()`。
找到 CPU 编号对应的 percpu 数组元素，`void *array_element = array_ptr + section_size * cpu_curr;`。
找到 percpu 变量的地址偏移，`int offset = (uint64_t)&per_cat - (uint64_t)&per_begin;`。
找到 percpu 变量的地址，`void *cat_addr = array_element + offset;`。
获得 percpu 变量指针，`(struct cat_t *)cat_addr;`。
之后就可以用变量指针来读写变量。