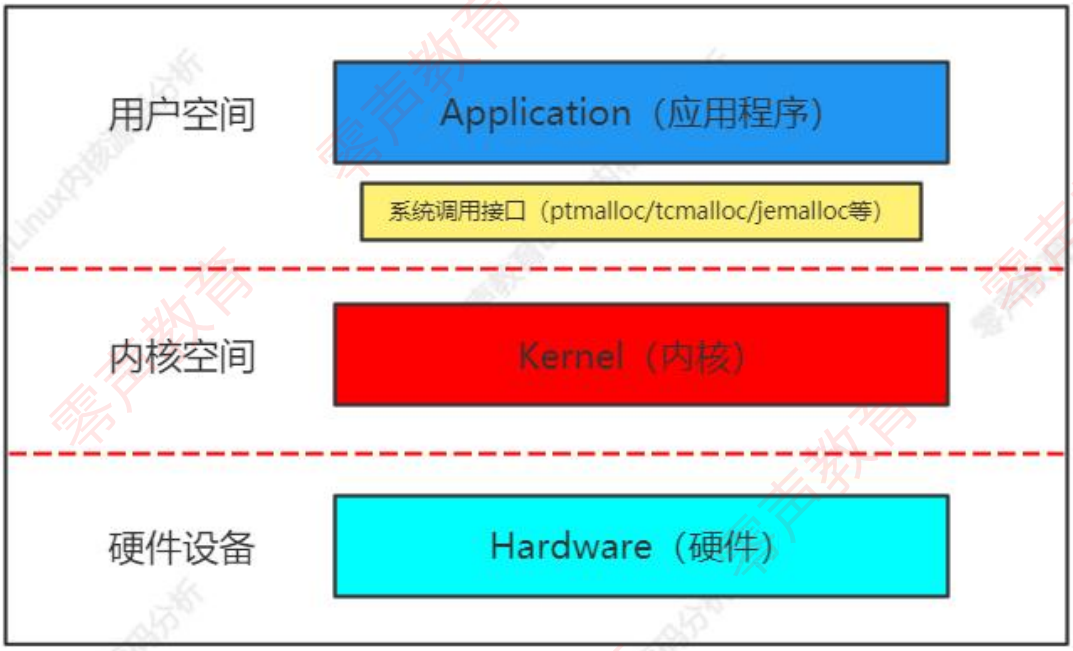
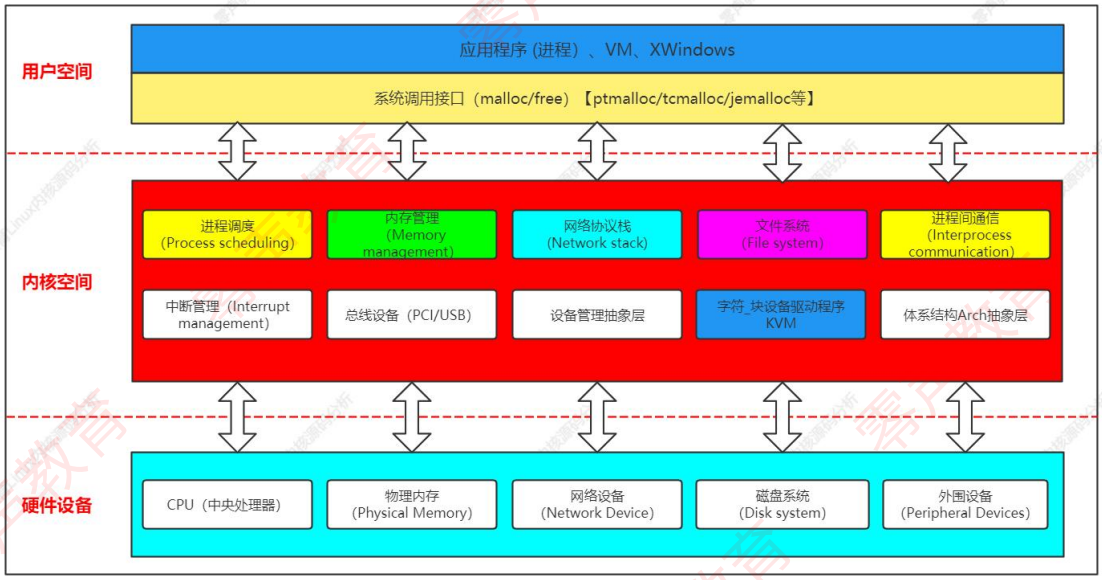
第 001 讲 Linux 内核源码分析（进程管理专题）

Linux 内核源码分析架构图





一、进程原理分析

1、进程基础知识

Linux 内核把进程称为任务(task)，进程的虚拟地址空间分

为用户虚拟地址空间和内核虚拟地址空间，所有进程共享内核虚拟地址空间，每个进程有独立的用户空间虚拟地址空间。

进程有两种特殊形式：没有用户虚拟地址空间的进程称为内核线程，共享用户虚拟地址空间的进程称为用户线程。通用在不会引起混淆的情况下把用户线程简称为线程。共享同一个用户虚拟地址空间的所有用户线程组成一个线程组。

C 标准库的进程专业术语和 Linux 内核的进程专业术语对应关系如下：

C 标准库的进程专业术语	Linux 内核的进程专业术语
包含多个线程的进程	线程组
只有一个线程的进程	进程或任务
线程	共享用户虚拟地址空间的进程

2、Linux 进程四要素

a. 有一段程序供其执行；

b. 有进程专用的系统堆栈空间；

c. 在内核有 task_struct 数据结构；

d. 有独立的存储空间，拥有专有的用户空间。

如果只具备前三条而缺少第四条，则称为“线程”。如果完全没有用户空间，就称为“内核线程”；而如果共享用户空间映射就称为“用户线程”。内核为每个进程分配一个 task_struct 结构

体。实际分配两个连续物理页面(8192 字节)，数据结构 task_struct 的大小约占 1kb 字节左右，进程的系统空间堆栈的大小约为 7kb 字节（不能扩展，是静态确定的）。

3、进程描述符 task_struct 数据结构主要成员内核源码分析

include > linux > C sched.h > task_struct

```
592  struct task_struct {
593  #ifdef CONFIG_THREAD_INFO_IN_TASK
594      /*
595       * For reasons of header soup (see current_thread_info()), this
596       * must be the first element of task_struct.
597       */
598      struct thread_info      thread_info;
599  #endif
600      /* -1 unrunnable, 0 runnable, >0 stopped: */
601      volatile long           state;
602
```

4、创建新进程分析

在 Linux 内核中，新进程是从一个已经存在的进程复制出来的，内核使用静态数据结构造出 0 号内核线程，0 号内核线程分叉生成 1 号内核线程和 2 号内核线程（kthreadd 线程）。1 号内核线程完成初始化以后装载用户程序，变成 1 号进程，其他进程都是 1 号进程或者它的子孙进程分叉生成的；其他内核线程是 kthreadd 线程分叉生成的。

Linux 3 个系统调用创建新的进程：

a. fork(分叉)：子进程是父进程的一个副本，采用写时复制技术。

b. vfork：用于创建子进程，之后子进程立即调用 execve 以装

载新程序的情况，为了避免复制物理页，父进程会睡眠等待子进程装载新程序。现在 fork 采用了写时复制技术，vfork 失去了速度优势，已经被废弃。

c.clone（克隆）：可以精确地控制子进程和父进程共享哪些资源。这个系统调用的主要用处是可供 pthread 库用来创建线程。clone 是功能最齐全的函数，参数多使用复杂，fork 是 clone 的简化函数。

kernel > C fork.c > ...

```
2293
2294  #ifdef __ARCH_WANT_SYS_FORK
2295  SYSCALL_DEFINE0(fork)
2296  {
2297  #ifdef CONFIG_MMU
2298      return _do_fork(SIGCHLD, 0, 0, NULL, NULL, 0);
2299  #else
2300      /* can not support in nommu mode */
2301      return -EINVAL;
2302  #endif
2303  }
2304  #endif
2305
2306  #ifdef __ARCH_WANT_SYS_VFORK
2307  SYSCALL_DEFINE0(vfork)
2308  {
2309      return _do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, 0,
2310              0, NULL, NULL, 0);
2311  }
2312  #endif
2313
```

kernel > C fork.c > ...

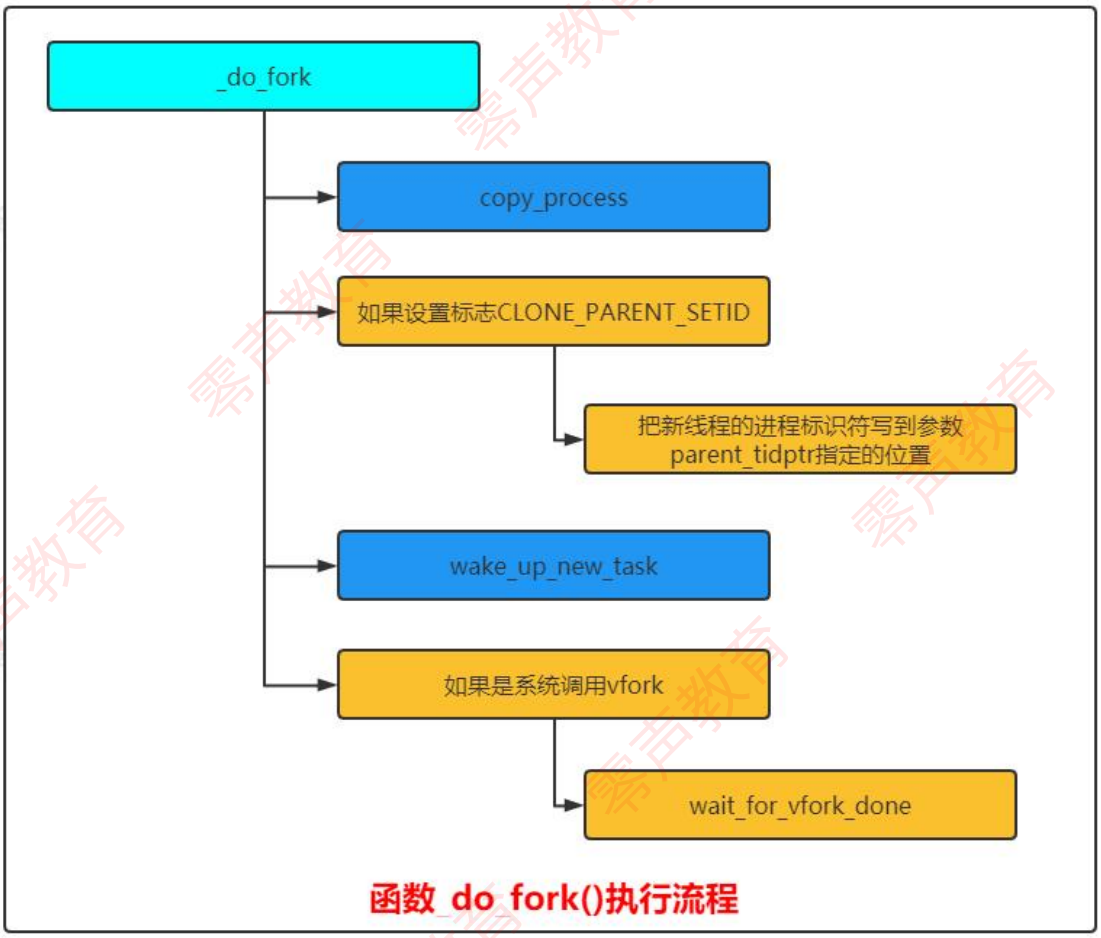
```
2319  #ifdef __ARCH_WANT_SYS_CLONE
2320  #ifdef CONFIG_CLONE_BACKWARDS
2321  SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
2322          int __user *, parent_tidptr,
2323          unsigned long, tls,
2324          int __user *, child_tidptr)
2325  #elif defined(CONFIG_CLONE_BACKWARDS2)
2326  SYSCALL_DEFINE5(clone, unsigned long, newsp, unsigned long, clone_flags,
2327          int __user *, parent_tidptr,
2328          int __user *, child_tidptr,
2329          unsigned long, tls)
2330  #elif defined(CONFIG_CLONE_BACKWARDS3)
2331  SYSCALL_DEFINE6(clone, unsigned long, clone_flags, unsigned long, newsp,
2332          int, stack_size,
2333          int __user *, parent_tidptr,
2334          int __user *, child_tidptr,
2335          unsigned long, tls)
2336  #else
2337  SYSCALL_DEFINE5(clone, unsigned long, clone_flags, unsigned long, newsp,
2338          int __user *, parent_tidptr,
2339          int __user *, child_tidptr,
2340          unsigned long, tls)
2341  #endif
2342  {
2343      return _do_fork(clone_flags, newsp, 0, parent_tidptr, child_tidptr, tls);
2344  }
2345  #endif
```

Linux 内核定义系统调用的独特方式，目前以系统调用 fork 为例：创建新进程的 3 个系统调用在文件"kernel/fork.c"中，它们把工作委托给函数_do_fork。具体源码分析如下：

kernel > C fork.c

```
2194  /*
2195   *  Ok, this is the main fork-routine.
2196   *
2197   * It copies the process, and if successful kick-starts
2198   * it and waits for it to finish using the VM if required.
2199   */
2200  long _do_fork(unsigned long clone_flags,
2201                unsigned long stack_start,
2202                unsigned long stack_size,
2203                int __user *parent_tidptr,
2204                int __user *child_tidptr,
2205                unsigned long tls)
2206  {
2207      struct completion vfork;
2208      struct pid *pid;
2209      struct task_struct *p;
2210      int trace = 0;
2211      long nr;
```

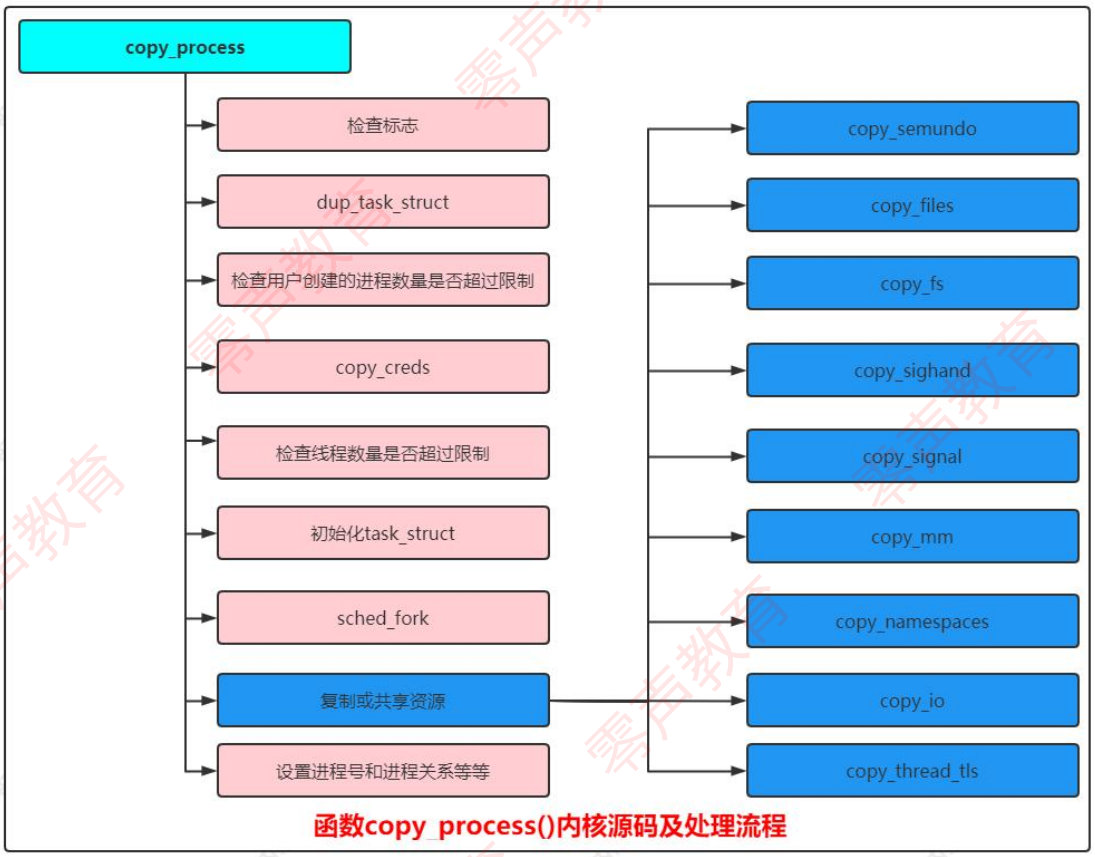
Linux 内核函数_do_fork()执行流程如下图所示：



具体核心处理函数为 copy_process()内核源码如下：

```
kernel > C fork.c > copy_process(unsigned long, unsigned long, unsigned long, int __user *, pid *, int, unsigned long, int)
1665
1666  /*
1667   * This creates a new process as a copy of the old one,
1668   * but does not actually start it yet.
1669   *
1670   * It copies the registers, and all the appropriate
1671   * parts of the process environment (as per the clone
1672   * flags). The actual kick-off is left to the caller.
1673   */
1674  static __latent_entropy struct task_struct *copy_process(
1675                  unsigned long clone_flags,
1676                  unsigned long stack_start,
1677                  unsigned long stack_size,
1678                  int __user *child_tidptr,
1679                  struct pid *pid,
1680                  int trace,
1681                  unsigned long tls,
1682                  int node)
1683  {
1684      int retval;
1685      struct task_struct *p;
1686      struct multiprocess_signals delayed;
1687
```

函数 copy_process()：创建新进程的主要工作由此函数完成，具体处理流程如下图所示：

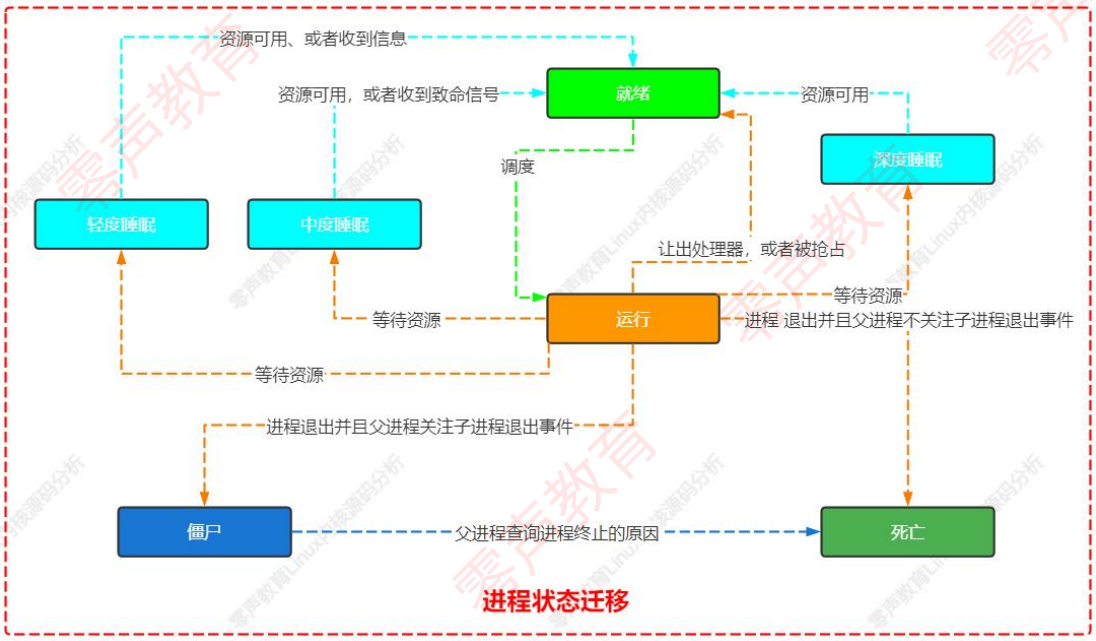


同一个线程组的所有线程必须属于相同的用户命名空间和进程号命

名空间。

二、剖析进程状态迁移

进程主要有 7 种状态：就绪状态、运行状态、轻度睡眠、中度睡眠、深度睡眠、僵尸状态、死亡状态，它们之间状态变迁如下：



三、内核调度策略及优先级

1、Linux 内核支持调度策略

先进先出调度（SCHED_FIFO）、轮流调度（SCHED_RR）、限期调度策略（SCHED_DEADLINE)采用不同的调度策略调度实时进程。

普通进程支持两种调度策略：标准轮流分时（SCHED_NORMAL）和 SCHED_BATCH 调度普通的非实时进程。

空闲（SCHED_IDLE）则在系统空闲时调用 idle 进程。

指量调度策略（SCHED_BATCH），Linux 内核引入完全公平调度算法之后。限期调度策略必须 3 个参数：运行时间 runtime、截止期限 deadline、周期 period。每一个周期运行一次，在截止期限之前执行完，一次运行的时间长度是 runtime。

标准轮流分时策略使用完全公平调度算法（把处理器时间公平地分配给每个进程）。

2、进程优先级

限期进程的优先级比实时进程高，实时进程的优先级比普通进程高。

限制进程的优先级是-1。

实时进程的实时优先级是 1-99，优先级数值越大，表示优先级越高。

普通进程的静态优先级是 100-139，优先级数值越小，表示优先级越高，可通过修改 nice 值改变普通进程的优先级，优先级等于 120 加上 nice 值。

在 task_struct 结构体中，4 个成员和优先级有关如下：

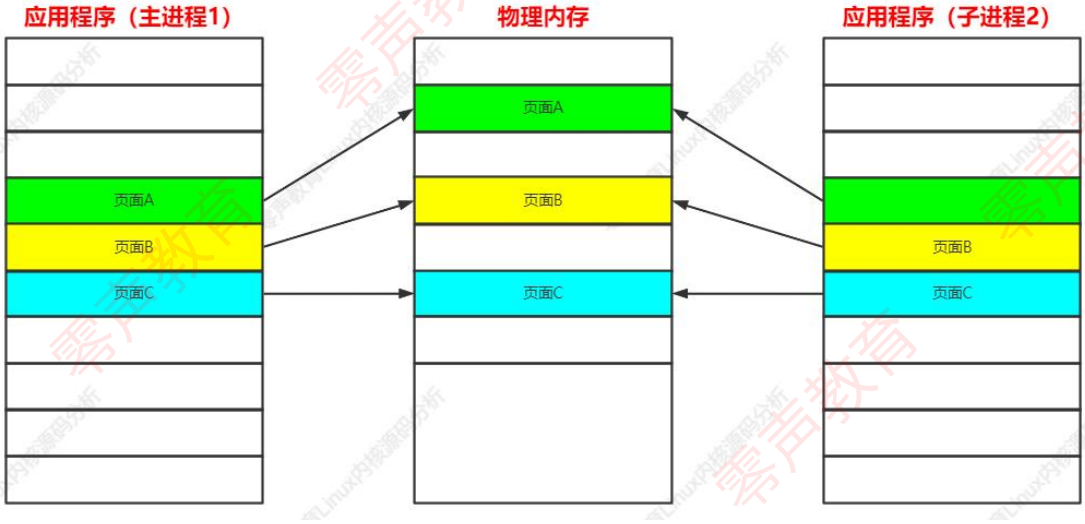
include > linux > C sched.h > task_struct > static_prio

```
637
638     int             prio;
639     int             static_prio;
640     int             normal_prio;
641     unsigned int            rt_priority;
642
```

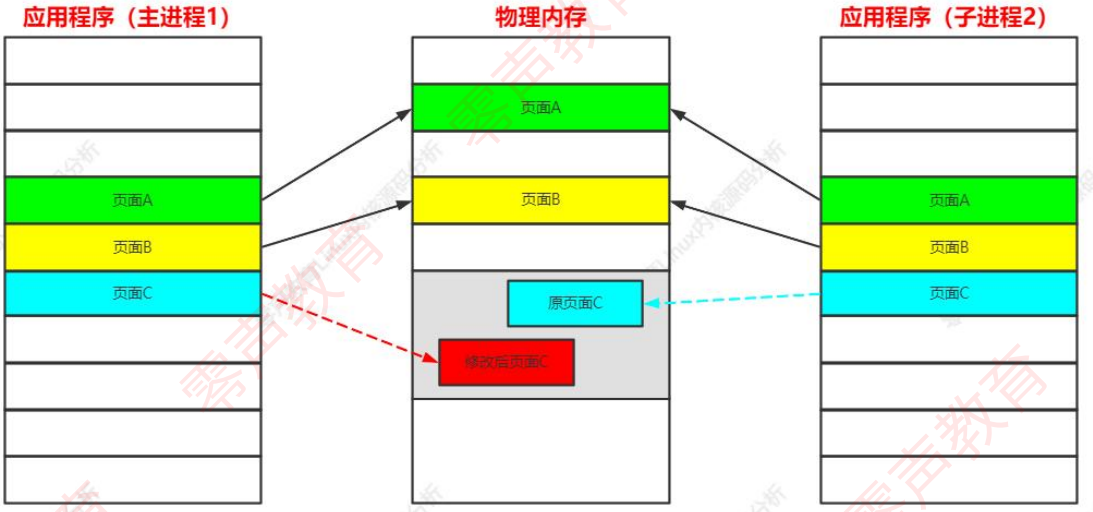
四、写时复制技术

写时复制核心思想：只有在不得不复制数据内容时才去复制数据内容。

应用程序（进程 1）修改页面 C 之前：



应用程序（进程 1）修改页面 C 之后：



备注：只有可修改的页面才需要标记为写时复制，不能修改的页面可以由父进程和子进程共享。

五、调度器分析及系统调用实现

1、核心调度器

调度器的实现基于两个函数：周期性调度器函数和主调度器函数。这些函数根据现有进程的优先级分配 CPU 时间。这也是为什么整个方法称之为优先调度的原因。

a.周期性调度器函数

周期性调度器在 scheduler_tick 中实现，如果系统正在活动中，内核会按照频率 HZ 自动调用该函数。该函数主要有两个任务如下：

(1) 更新相关统计量：管理内核中与整个系统和各个进程的调度相关的统计量。其间执行的主要操作是对各种计数器加 1。

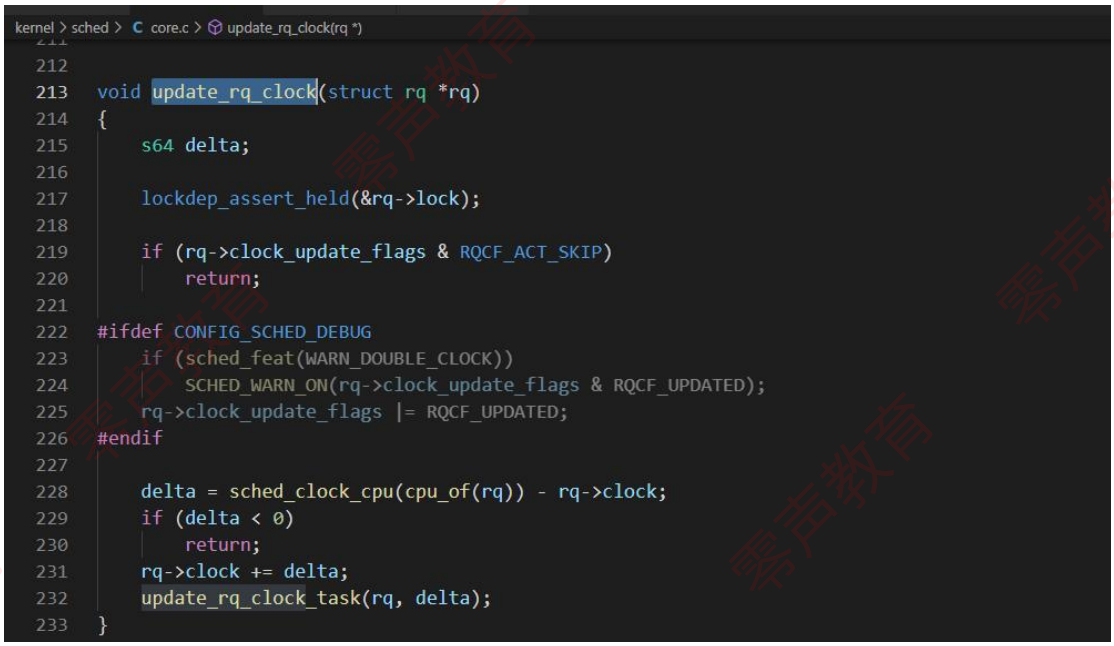
(2) 激活负责当前进程的调度类的周期性调度方法。

kernel > sched > C core.c > ...

```
3089  /*
3090   * This function gets called by the timer code, with HZ frequency.
3091   * We call it with interrupts disabled.
3092   */
3093  void scheduler_tick(void)
3094  {
3095      int cpu = smp_processor_id();
3096      struct rq *rq = cpu_rq(cpu);
3097      struct task_struct *curr = rq->curr;
3098      struct rq_flags rf;
3099
3100      sched_clock_tick();
3101
3102      rq_lock(rq, &rf);
3103
3104      update_rq_clock(rq);
3105      curr->sched_class->task_tick(rq, curr, 0);
3106      cpu_load_update_active(rq);
3107      calc_global_load_tick(rq);
3108
3109      rq_unlock(rq, &rf);
3110
3111      perf_event_task_tick();
3112
```

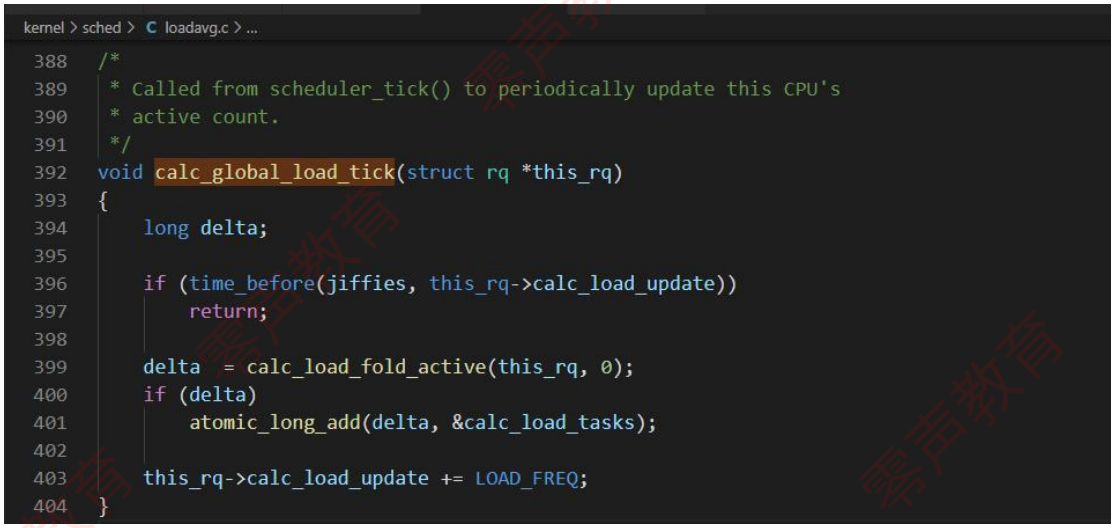
更新统计量函数：update_rq_clock()/calc_global_load_tick()

<update_rq_clock 函数>



```
kernel > sched > C core.c > update_rq_clock(rq *)
212
213  void update_rq_clock(struct rq *rq)
214  {
215      s64 delta;
216
217      lockdep_assert_held(&rq->lock);
218
219      if (rq->clock_update_flags & RQCF_ACT_SKIP)
220          return;
221
222  #ifdef CONFIG_SCHED_DEBUG
223      if (sched_feat(WARN_DOUBLE_CLOCK))
224          SCHED_WARN_ON(rq->clock_update_flags & RQCF_UPDATED);
225      rq->clock_update_flags |= RQCF_UPDATED;
226  #endif
227
228      delta = sched_clock_cpu(cpu_of(rq)) - rq->clock;
229      if (delta < 0)
230          return;
231      rq->clock += delta;
232      update_rq_clock_task(rq, delta);
233  }
```

<calc_global_load_tick 函数>



```
kernel > sched > C loadavg.c > ...
388  /*
389   * Called from scheduler_tick() to periodically update this CPU's
390   * active count.
391   */
392  void calc_global_load_tick(struct rq *this_rq)
393  {
394      long delta;
395
396      if (time_before(jiffies, this_rq->calc_load_update))
397          return;
398
399      delta  = calc_load_fold_active(this_rq, 0);
400      if (delta)
401          atomic_long_add(delta, &calc_load_tasks);
402
403      this_rq->calc_load_update += LOAD_FREQ;
404  }
```

b. 主调度器函数

在内核中的许多地方，如果要将 CPU 分配给与当前活动进程不同的另一个进程，都会直接调用主调度器函数（schedule）。

```
kernel > sched > C core.c > schedule(void)
3495  asmlinkage __visible void __sched schedule(void)
3496  {
3497      struct task_struct *tsk = current;
3498
3499      sched_submit_work(tsk);
3500      do {
3501          preempt_disable();
3502          __schedule(false);
3503          sched_preempt_enable_no_resched();
3504      } while (need_resched());
3505  }
3506  EXPORT_SYMBOL(schedule);
3507
```

主调度器负责将 CPU 的使用权从一个进程切换到另一个进程。周期性调度器只是定时更新调度相关的统计信息。cfs 队列实际上是用红黑树组织的，rt 队列是用链表组织的。

2、调度类及运行队列

a. 调度类

为方便添加新的调度策略，Linux 内核抽象一个调度类 sched_class，目前为止实现 5 种调度类：

调度类	调度策略	调度算法	调度对象	task_tick函数定义
stop_sched_class（停机调度类）	无	无	停机进程	task_tick_stop
dl_sched_class（限期调度类）	SCHED_DEADLINE	最早期限优先	限期进程	task_tick_dl
rt_sched_class（实时调度类）	SCHED_FIFO	先进先出	实时进程	task_tick_rt
	SCHED_RR	轮流调度		
fair_sched_class（公平调度类）	SCHED_NORMAL	完全公平调度算法	普通进程	task_tick_fair
	SCHED_IDLE			
idle_sched_class（空闲调度类）	无	无	每个处理器上的空闲线程	task_tick_idle

调度类优先级从高到低排序：停机调度类->限期调度类->实时调度类->公平调度类和空闲调度类。

公开调度类使用完全公平调度算法（引入虚拟运行时间这个东西）？

虚拟运行时间=实际运行时间*nice0 对应的权重/进程的权重。

进程的时间片=（调度周期*进程的权重/运行队列中所有进程的权重之和）

b.运行队列

每个处理器有一个运行队列，结构体是 rq，定义的全局变量如下：

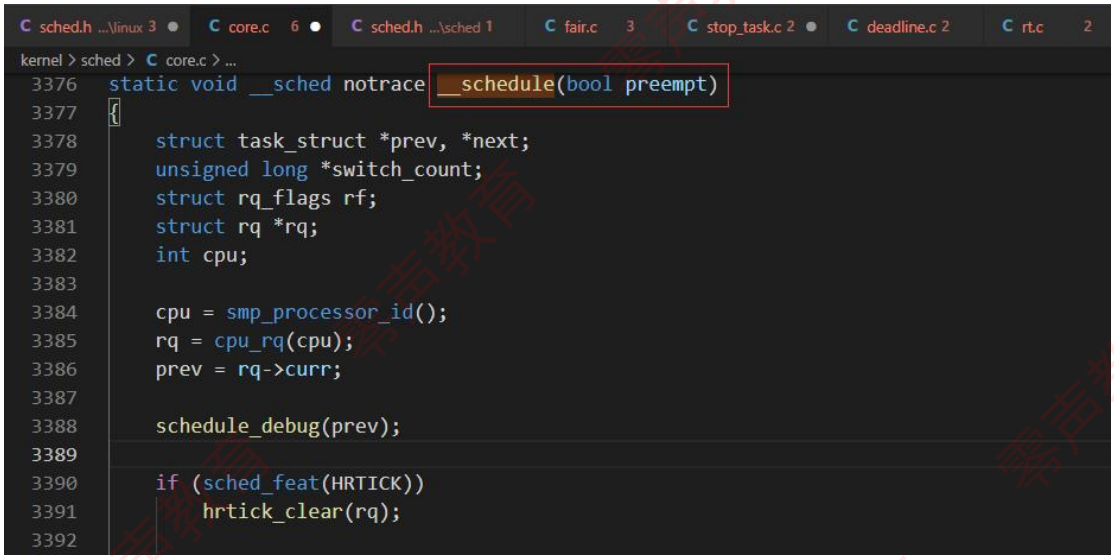
```
kernel > sched > C core.c > DEFINE_PER_CPU_SHARED_ALIGNED(rq, runqueues)
39  #define CREATE_TRACE_POINTS
40  #include <trace/events/sched.h>
41
42  DEFINE_PER_CPU_SHARED_ALIGNED(struct rq, runqueues);
43
```

rq 是描述就绪队列，其设计是为每一个 CPU 就绪队列，本地进程在本地队列上排序：

3、调度进程

主动调度进程的函数是 schedule() ，它会把主要工作委托给__schedule()去处理。

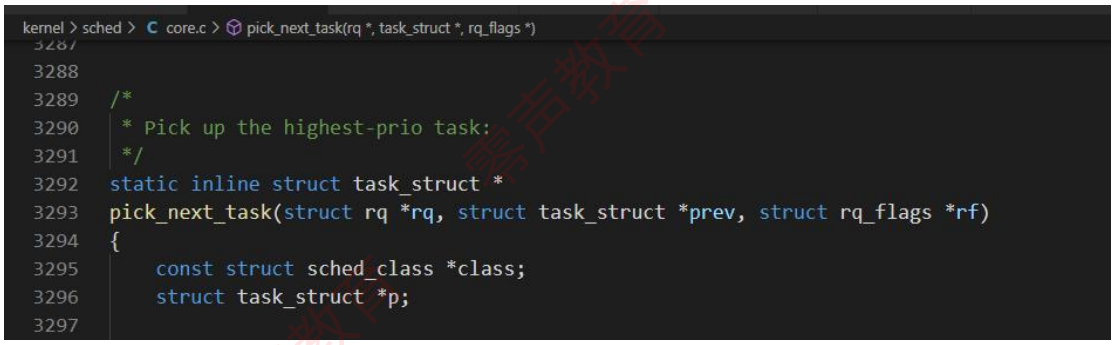
```
C sched.h ...\linux 1 ●  C core.c 6 ●  C sched.h ...\sched 1  C fair.c 3  C stop_task.c 2 ●  C deadline.c 2  C rt.c 2
kernel > sched > C core.c > ...
3504  asmlinkage __visible void __sched schedule(void)
3505  {
3506      struct task_struct *tsk = current;
3507
3508      sched_submit_work(tsk);
3509      do {
3510          preempt_disable();
3511          __schedule(false);
3512          sched_preempt_enable_no_resched();
3513      } while (need_resched());
3514  }
3515  EXPORT_SYMBOL(schedule);
3516
```

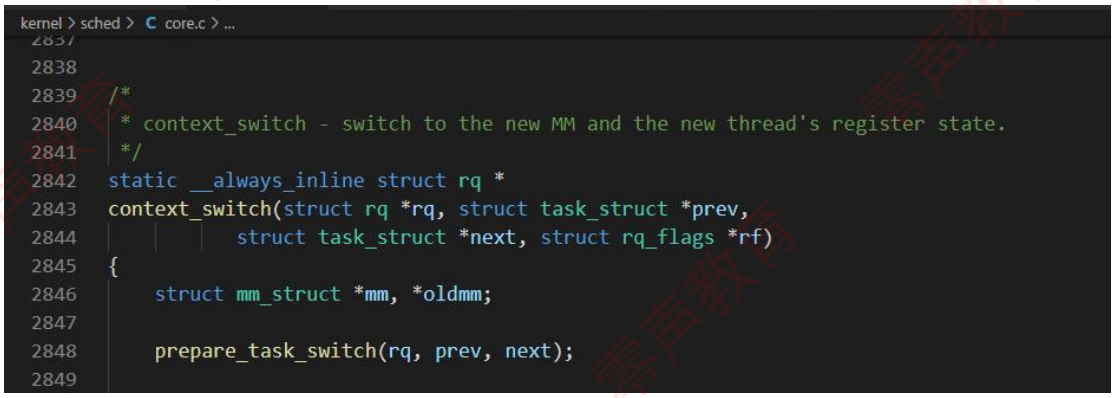


函数__shcedule 的主要处理过程如下：

调用 pick_next_task()以选择下一个进程。

调用 context_switch()以切换进程。





a、切换用户虚拟地址空间，ARM64 架构使用默认的

switch_mm_irqs_off，其内核源码定义如下：

include > linux > C mmu_context.h > ...

```
12
13  /* Architectures that care about IRQ state in switch_mm can override this. */
14  #ifndef switch_mm_irqs_off
15  # define switch_mm_irqs_off switch_mm
16  #endif
17
```

switch_mm 函数内核源码处理如下：

arch > arm64 > include > asm > C mmu_context.h > ...

```
207
208 static inline void
209 switch_mm(struct mm_struct *prev, struct mm_struct *next,
210           struct task_struct *tsk)
211 {
212     if (prev != next)
213         __switch_mm(next);
214
```

arch > arm64 > include > asm > C mmu_context.h > ...

```
191
192 static inline void __switch_mm(struct mm_struct *next)
193 {
194     unsigned int cpu = smp_processor_id();
195
196     /*
197      * init_mm.pgd does not contain any user mappings and it is always
198      * active for kernel addresses in TTBR1. Just set the reserved TTBR0.
199      */
200     if (next == &init_mm) {
201         cpu_set_reserved_ttbr0();
202         return;
203     }
204
205     check_and_switch_context(next, cpu);
206 }
```

b、切换寄存器，宏 switch_to 把这项工作委托给函数 __switch_to：

include > asm-generic > C switch_to.h

```
24
25  #define switch_to(prev, next, last)                 \
26      do {                                            \
27          ((last) = __switch_to((prev), (next)));     \
28      } while (0)
29
30  #endif /* __ASM_GENERIC_SWITCH_TO_H */
31
```

```
arch > arm64 > kernel > C process.c > ...
339
340  /*
341   * Thread switching.
342   */
343  __notrace_funcgraph struct task_struct *__switch_to(struct task_struct *prev,
344                  struct task_struct *next)
345  {
346      struct task_struct *last;
347
348      fpsimd_thread_switch(next);
349      tls_thread_switch(next);
350      hw_breakpoint_thread_switch(next);
351      contextidr_thread_switch(next);
352      entry_task_switch(next);
353      uao_thread_switch(next);
354
355      /*
356       * Complete any pending TLB or cache maintenance on this CPU in case
357       * the thread migrates to a different CPU.
358       */
359      dsb(ish);
360
361      /* the actual thread switch */
362      last = cpu_switch_to(prev, next);
363
364      return last;
365  }
```

4、调度时机

调度进程的时机如下：

进程主动调用 schedule()函数。

周期性地调度，抢占当前进程，强迫当前进程让出处理器。

唤醒进程的时候，被唤醒的进程可能抢占当前进程。

创建新进程的时候，新进程可能抢占当前进程。

如果我们编译内核时开启对内核抢占的支持，那么内核含增加一些指占点。

a、主动调度

进程在用户模式下运行的时候，无法直接调用 schedule()函数，只能通过系统调用进入内核模式，如果系统调用需要等待某个

资源，如互斥锁或信号量，就会把进程的状态设置为睡眠状态，然后调用 schedule()函数来调度进程。

进程也可以通过系统调用 shced_yield()让出处理器，这种情况下进程不会睡眠。

在内核中有 3 种主动调度方式：

直接调用 schedule()函数来调用进程。

调用有条件重调度函数 cond_resched()。

如果需要等待某个资源。

b、周期调度

有些“地痞流氓”进程不主动让出处理器，内核只能依靠周期性的时钟中断夺回处理器的控制权，时钟中断是调度器的脉博。时钟中断处理程序检查当前进程的执行时间有没有超过限额，如果超过限额，设置需要重新调度的标志。当时钟中断处理程序准备返点处理器还给被打断的进程时，如果被打断的进程在用户模式下运行，就检查有没有设置需要重新调度的标志，如果设置了，调用 schedule 函数以调度进程。

如果需要重新调度，就为当前进程的 thread_info 结构体的成员 flags 设置需要重新调度的标志。

5、SMP 调度

在 SMP 系统中，进程调度器必须支持如下：

需要使用每个处理器的负载尽可能均衡。

可以设置进程的处理器亲和性，即允许进程在哪些处理器上执行。

可以把进程从一个处理器迁移到另一个处理器。

a、进程的处理器亲和性

设置进程的处理器亲和性，通俗就是把进程绑定到某些处理器，只允许进程在某些处理器上执行，默认情况是进程可以在所有处理器上执行。应用编程接口和使用 cpuset 配置具体详解分析。

b、限期调度类的处理器负载均衡

限期调度类的处理器负载均衡简单，调度选择下一个限期进程的时候，如果当前正在执行的进程是限期进程，将会试图从限期进程超载的处理器把限期进程搞过来。

限期进程超载定义：

限期运行队列至少有两个限期进程。

至少有一个限期进程绑定到多个处理器。

kernel > sched > C deadline.c > pull_dl_task(rq *)

```
1579  static void pull_dl_task(struct rq *this_rq)
1580  {
1581      int this_cpu = this_rq->cpu, cpu;
1582      struct task_struct *p;
1583      bool resched = false;
1584      struct rq *src_rq;
1585      u64 dmin = LONG_MAX;
1586
1587      if (likely(!dl_overloaded(this_rq)))
1588          return;
1589
```

c、实时调度类的处理器负载均衡

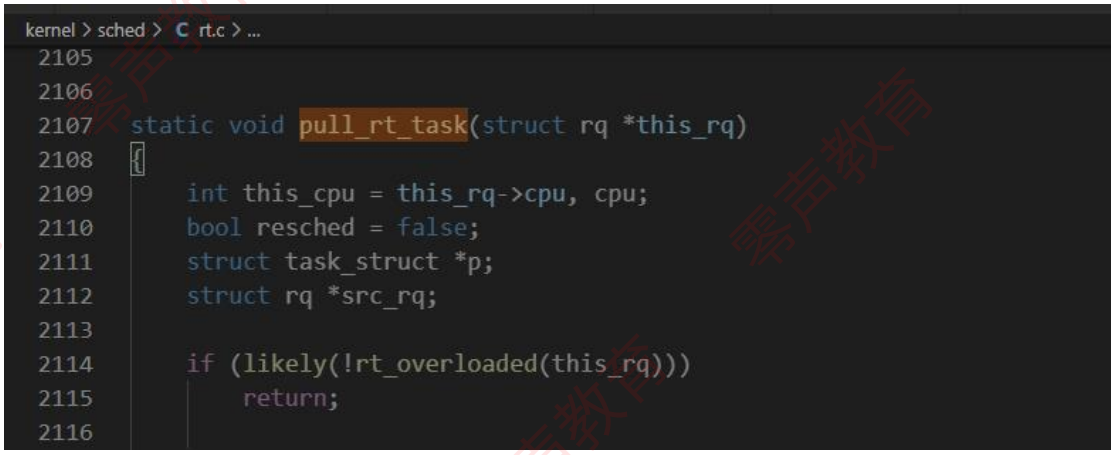
实时调度类的处理器负载均衡和限期调度类相似。调度器选择下一

个实时进程时，如果当前处理器的实时运行队列中的进程的最高调度优先级比当前正在执行的进程的调度优先级低，将会试图从实时进程超载的处理器把可推送实时进程拉过来。

实时进程超载的定义：

实时运行队列至少有两个实时进程。

至少有一个可推送实时进程。



d、公平调度类的处理器负载均衡

目前多处理器系统有两种体系结构：NUMA 和 SMP。

处理器内部的拓扑如下：

a. 核(core)：一个处理器包含多个核，每个核独立的一级缓存，所有核共享二级缓存。

b. 硬件线程：也称为逻辑处理器或者虚拟处理器，一个处理器或者核包含多个硬件线程，硬件线程共享一级缓存和二级缓存。MIPS 处理器的叫法是同步多线程(Simultaneous Multi-Threading,SMT)，英特尔对它的称为超线程。

Linux 内核技术常见面试题：

1、为什么自旋锁的临界区不允许发生抢占？

2、自述 MCS 锁机制的实现原理？

3、PG_locked 常见使用方法？

4、softlockup 和 hardlockup ？

5、问的 c++ lamda 怎么递包，怎么判断内存溢出？

零声教育