preempt rt 优化内容

Rt_mutex

- 解决了优先级反转
- 大部分内核自旋锁被转换为rt_mutex并且实现了优先级继承, 这种技术要求低优先级任务继承它们共享的资源上任何高优先级任务的优先级。一旦高优先级的任务开始等待，这个优先级的改变就应该发生，应该在资源被释放时结束。
- 信号量没有所有者的概念，没有优先级继承
- 在struct rt_mutex中记录了owner和所有的waiters（进程），使用红黑树实现，称为Mutex Waiter List，优先级最高的那一个叫top waiter。

  struct rt_mutex_waiter是用来链接rt_mutex和task_struct的数据结构，保存了waiter是哪个进程（task_struct）在等待哪个lock（rt_mutex）

  struct task_struct中同样有一个waiters列表，使用红黑树实现，称为Task PI List列表中包含此进程持有所有rt_mutex的top waiter，称作top pi waiter。
- rt_mutex对外呈现为mutex，仍然需要保护的自旋锁对外表现为raw_spin_lock

High resolution timers

- hr times架构允许使用 硬件计时器 在正确的时刻执行中断，精度比jiffies更高
- nanosleep,itimers, posix timers都可以在正常linux内核实现hr timers
- 当实时抢占补丁中，hr timer硬中断环境中，itimer和posix interval timer都不能到期时传递信号。
- nanosleep函数不受影响，因为计时器到期时的唤醒函数是在高分辨率计时器中断的上下文中执行的。 如果应用程序没有使用异步信号处理程序，建议使用设置了TIMER_ABSTIME标志的clock_nanosleep()函数，而不是等待定期计时器信号传递。 应用程序必须维护下一个时间间隔本身的绝对到期时间，具体操作为添加和规范化两个timespec结构体。 这样做的好处是极大地降低了最大延迟(~50us)和操作系统开销。

Thread Interrupt handler

- linux中断服务例程(ISR)在硬中断上下文中执行，执行同时使硬件中断失效，这意味着软中断和抢占失效。在ISR上下文中执行的中断处理程序也禁用了硬中断。linux中软中断不会抢占另一个软中断，并且被抢占的软中断不能睡眠。
- preempt_rt补丁中，几乎所有中断处理程序都是线程
- 线程化的中断处理程序的调度策略被设置为 SCHED_FIFO，并且默认优先级为50。

RCU（Read-Copy Update）

- RCU机制：先分配一段新的内存空间，创建一个旧数据的copy，然后writer更新这个copy，修改完成后writer将这个更新发布，发布之后开始读取操作的reader获得新信息，等所有旧数据区的reader都完成了相关操作，writer释放旧数据内存区域。
- RCU在软中断上下文中执行并且默认非抢占

- RCU的回调：writer需要等待所有正在读的reader读取完成后，将更新发布。如果读取时间过长，会让writer干等。所以使用基于回调机制的call_rcu()来替换更新发布，传递进去的函数是更新发布函数（实际上功能是释放掉旧指针指向的内存空间）
- reader，调用rcu_read_lock()进入临界区后，因为所使用的节点可能被updater（见下writer）解除引用，所以需要通过rcu_dereference()保留一份对这个节点的指针指向。接下来这些reader对该结点的操作都是引用这个临时指针，在reader推出临界区之前只能使用旧数据。
- writer被划分为updater和reclaimer
    - updater更新copy的数据，通过rcu_assign_pointer()，用这个copy替换原节点在链表中的位置，并移除对原节点的引用，然后调用synchronize_rcu()或call_rcu()进入grace period（见下）。因为synchronize_rcu()会阻塞等待，所以只能在进程上下文中使用，而call_rcu()可在中断上下文中使用。
    - 在最后一个引用节点的reader退出临界区后，reclaimer会释放这个节点。
- grace period的生命周期
    - 开始：由作为writer的CPU发起，发起的时间是writer调用synchronize_rcu()或call_rcu()，准备释放一个对象的时候。释放的具体操作（callback）被填入这个对象内含的rcu_head中，然后被rcu_head带着加入到待执行的链表中。
    - 结束：在不可抢占的RCU中，reader进入临界区时，reader所在的CPU上的调度是关闭的，直到退出临界区后，调度才会重新打开。并且临界区内的代码是被要求不能睡眠/阻塞的，因此不会发生线程切换。
- CONFIG_PREEMPT_RCU=y设置RCU读可抢占。
- RCU callback offloading优化选项
    - 主线linux中，RCU回调在软中断上下文中调用，这种回调通常会释放内存，因此内存分配器使用slowpath时会造成较大延迟。可以在指定的cpu上使用RCU callback offloading来提高效率，每个用offloaded回调的CPU都会有一组rcuo kthread。这些kthread可以被分配给特定的CPU，并根据需要分配优先级。
    - 使用RCU回调offloading意味着call_rcu()方法会因为原子操作、缓存丢失、唤醒等问题带来更大的开销，这个唤醒开销可以通过rcu_nocb_poll内核引导函数从调用call_rcu()的任务转移到rcuo kthread，但是代价时轮询唤醒会降低能源效率。
- RCU Priority Boosting优先级提升
    - 可抢占RCU的缺点是，低优先级的任务可能会在RCU读临界区中间被抢占，如果系统高优先级任务一直抢占，那么低优先级任务可能永远不会恢复，也不可能离开其临界区，这导致了RCU grace period无限延长。这表明事件驱动的实时应用程序应该留出大量空闲时间，来允许低优先级任务继续进行。
    - **一种解决方式**：使用CONFIG_RCU_BOOST=y，默认情况下将阻塞当前grace period超过半秒的任务提升到实时优先级1。这在PREEMPT RT补丁中被设为默认。

printk

- 为每个console设立一个线程来完成打印到console操作

---

cyclictest测量orin上打preempt RT补丁后linux实时性

cyclictest通过起pthread线程测量时延来检测实时性。测量的延时为：中断延时+调度延时，即从产生时钟中断（hrtimer）开始，进入ISR，唤醒实时进程，推出中断，进程加入runqueue等待，直到进程获得执行这段时间

单处理器

- 单处理器单线程 利用率0% 时延（ns）

```
lotus@tegra-ubuntu:~$ sudo cyclictest -t 1  -p 80 -i 10000 -d 0 -N -a 3 --quiet
# /dev/cpu_dma_latency set to 0us
^CT: 0 (997157) P:80 I:10000 C:    263 Min:   3232 Act:    4448 Avg:    4065 Max:   13152
```

平均：4us 最大：13us

- 单处理器起2000线程 利用率40%时延（ns）

```
T:1995 (991346) P:80 I:10000 C:  14466 Min:   2944 Act:    8480 Avg:    8205 Max: 1077760
T:1996 (991347) P:80 I:10000 C:  14466 Min:   2400 Act:    9728 Avg:    6290 Max: 1308768
T:1997 (991348) P:80 I:10000 C:  14466 Min:   2848 Act:    9152 Avg:    5351 Max: 1295264
T:1998 (991349) P:80 I:10000 C:  14466 Min:   7136 Act:   11488 Avg:   10696 Max: 1286688
T:1999 (991350) P:80 I:10000 C:  14466 Min:   3040 Act:   12192 Avg:    6789 Max: 1309504
```

平均：5us左右 最大：1.3ms

- 单处理器起2500线程 利用率60% 时延（ns）

```
T:2495 (935922) P:80 I:10000 C:  17907 Min:   3200 Act: 1243424 Avg:  103743 Max: 9930880
T:2496 (935923) P:80 I:10000 C:  17907 Min:   2496 Act: 1241696 Avg:  102709 Max: 9927456
T:2497 (935924) P:80 I:10000 C:  17907 Min:   2560 Act: 1242080 Avg:  102346 Max: 9926560
T:2498 (935925) P:80 I:10000 C:  17907 Min:   2464 Act: 1239296 Avg:  102188 Max: 9924832
T:2499 (935926) P:80 I:10000 C:  17907 Min:   2432 Act: 1242336 Avg:  102139 Max: 9924000
```

平均：0.1ms 最大：10ms

- 单处理器起2600线程 利用率70%时延（ns）

```
T:2594 (979603) P:80 I:10000 C:  30728 Min:   5056 Act:  344928 Avg:  364538 Max: 9994880
T:2595 (979604) P:80 I:10000 C:  30724 Min:   5856 Act:  458464 Avg:  363805 Max:10000640
T:2596 (979605) P:80 I:10000 C:  30728 Min:   4864 Act:  576448 Avg:  364059 Max: 9986848
T:2597 (979606) P:80 I:10000 C:  30728 Min:   4320 Act: 2154272 Avg:  367002 Max: 9995520
T:2598 (979607) P:80 I:10000 C:  30728 Min:   4576 Act: 2052672 Avg:  368156 Max: 9995840
T:2599 (979612) P:80 I:10000 C:  30715 Min:  15520 Act: 1978976 Avg:  377420 Max:10036960
```

平均：0.37ms 最大：10ms

- 单处理器起3000线程 利用率80%时延（ns）太卡了测不了

多处理器

- 5核处理器起2500线程 每个核利用率10% 时延（ns）

```
T:2495 (1004696) P:80 I:10000 C:  31249 Min:   3936 Act:    6144 Avg:    5129 Max:   12608
T:2496 (1004697) P:80 I:10000 C:  31249 Min:   3232 Act:    4992 Avg:    4169 Max:   20800
T:2497 (1004698) P:80 I:10000 C:  31249 Min:   3264 Act:    8864 Avg:    4352 Max:   14496
T:2498 (1004699) P:80 I:10000 C:  31249 Min:   3552 Act:    4256 Avg:    4542 Max:   15232
T:2499 (1004700) P:80 I:10000 C:  31249 Min:   3584 Act:    4736 Avg:    4654 Max:   11136
```

平均：4us 最大：10~20us

- 5核处理器起5000线程 每个核利用率40%~50% 时延（ns）

```
T:4995 (1058604) P:80 I:10000 C:  10002 Min:   4192 Act:   10656 Avg:    5512 Max:   12416
T:4996 (1058605) P:80 I:10000 C:  10002 Min:   4640 Act:   12928 Avg:    7261 Max:   17344
T:4997 (1058606) P:80 I:10000 C:  10002 Min:   3552 Act:    5408 Avg:    5085 Max:    7488
T:4998 (1058607) P:80 I:10000 C:  10002 Min:   3008 Act:    4448 Avg:    4812 Max:   15904
T:4999 (1058608) P:80 I:10000 C:  10002 Min:   3296 Act:    5568 Avg:    4747 Max:   20384
```

平均：5~10us 最大：20us左右

- 5核处理器起9000线程 每个核利用率70%~80% 时延（ns）

```
T:8994 (1158810) P:80 I:10000 C:   4457 Min:   7296 Act:    9120 Avg:    9206 Max:   23840
T:8995 (1158811) P:80 I:10000 C:   4457 Min:   3456 Act:    5120 Avg:    5034 Max:    7296
T:8996 (1158812) P:80 I:10000 C:   4457 Min:   7712 Act:   17856 Avg:   10030 Max:   17856
T:8997 (1158813) P:80 I:10000 C:   4457 Min:   3680 Act:   17824 Avg:    4974 Max:   17824
T:8998 (1158814) P:80 I:10000 C:   4456 Min:   8928 Act:   12320 Avg:   11214 Max:   24320
T:8999 (1158815) P:80 I:10000 C:   4457 Min:  12896 Act:   25600 Avg:   14880 Max:   41536
```

平均：10us左右 最大：20us左右，最多达到了近300us

- 10核处理器起500线程 每个核利用率10% 时延（ns）大量的时间花费在硬件中断上

```
T:495 (1167564) P:99 I:1000 C: 361056 Min:   2656 Act:    9664 Avg:    3860 Max:   13600
T:496 (1167565) P:99 I:1000 C: 361052 Min:   4384 Act:   13344 Avg:    5829 Max:   36384
T:497 (1167566) P:99 I:1000 C: 361049 Min:   2144 Act:   11360 Avg:    3792 Max:   11776
T:498 (1167567) P:99 I:1000 C: 361045 Min:   2816 Act:    4128 Avg:    3962 Max:   12544
T:499 (1167568) P:99 I:1000 C: 361041 Min:   2624 Act:    3168 Avg:    3475 Max:   20992
```

平均：5us左右 最大：20us左右

- 10核处理器起2000线程 每个核利用率70~80% 时延（ns）大量的时间花费在硬件中断上

```
T:1995 (1174107) P:99 I:1000 C:  10786 Min:   3520 Act:  363424 Avg:  405498 Max: 1697376
T:1996 (1174108) P:99 I:1000 C:  10592 Min:   2176 Act:  358208 Avg:  403563 Max: 1687008
T:1997 (1174109) P:99 I:1000 C:  10656 Min:   2304 Act:  522720 Avg:  422649 Max: 1697088
T:1998 (1174110) P:99 I:1000 C:  10640 Min:   4672 Act:  483616 Avg:  445784 Max: 1704096
T:1999 (1174111) P:99 I:1000 C:  10476 Min:   2400 Act:   91232 Avg:  431145 Max: 1624416
```

平均：0.4ms 最大：1.7ms

- 10核处理器起2500线程 每个核利用率80~90% 时延（ns）大量的时间花费在硬件中断上

```
T:2495 (1180154) P:99 I:1000 C:  20555 Min:   4576 Act:  979040 Avg:  752063 Max: 2115648
T:2496 (1180155) P:99 I:1000 C:  21018 Min:   2016 Act:  566240 Avg:  732361 Max: 2251904
T:2497 (1180156) P:99 I:1000 C:  20083 Min:   2368 Act:  851968 Avg:  785530 Max: 2308288
T:2498 (1180157) P:99 I:1000 C:  21024 Min:    896 Act:  258560 Avg:  751395 Max: 2231072
T:2499 (1180158) P:99 I:1000 C:  20947 Min:   1504 Act: 1074240 Avg:  733558 Max: 2180800
```

平均：0.7~0.8ms 最大：2.2ms

结论：启用核个数不需要太多，需要依据具体应用线程数考虑

---

pthread_create起线程过程

- pthread_create
  - pthread_create_2_1
    - 初始化，没有指定attr时用默认值
    - 继承父进程属性
    - atomic_increment(&_nptl_nthreads) 增加线程数
      - __arch_increment_body(LOCK_PREFIX, __arch, mem) -> 汇编
    - create_thread
      - ARCH_CLONE

- start_thread
  - ! not_first_call -> 起回调函数