基于 NuttX RTOS 的符合 POSIX 规范和 SMP 系统的高效任务调度器实现

202310558111463 铃芽小姐来死锁

杨翼飞[2] 陈冠一[1] 周桐欣[1] 赵帅[1, *] 黎卫兵[1, *]

[1] 中山大学 计算机学院（软件学院）
[2] 中山大学 软件工程学院
[*] 指导教师

中山大學
SUN YAT-SEN UNIVERSITY

目录

队伍简介

队员简介：

- 杨翼飞：中山大学软件工程学院软件工程专业 2020 级本科生，在操作系统和智能软件工程方向有一定的科研基础和实践经历，获得过相关比赛的一些奖项。
  - 个人主页：`https://yfyang.me/`
- 陈冠一，中山大学计算机学院计算机科学与技术专业 2020 级本科生，在操作系统和编译系统相关方向有一定项目经验和研究基础，有 ACM-ICPC 竞赛经验。
- 队员周桐欣，中山大学计算机学院计算机科学与技术专业 2022 级本科生，对嵌入式单片机开发有一定项目经验。

队伍简介

指导老师简介：

- 赵帅，硕士生导师，“百人计划”引进副教授。主要从事实时系统、操作系统领域的理论与应用研究，旨在围绕操作系统，为上层应用提供高性能、硬实时的计算与通讯保障，专注于复杂实时系统设计与分析、系统资源共享与管理、硬软件协同设计与寻优等具体方向。
- 黎卫兵，副教授，中山大学“百人计划”引进人才，从事医疗机器人、工业机器人、模块机器人（机器人方向涉及优化控制、视觉伺服、人机交互、自适应控制与学习、样机研制等）；神经网络（类脑计算、数值算法、深度学习）；运筹学与控制论相关方向的研究工作。

题目基本情况

- 题目编号：182
- 所属赛道：操作系统功能赛
- 题目名称：基于 NuttX RTOS 的符合 POSIX 规范和 SMP 系统的高效任务调度器实现
- 项目导师：黄齐 & 小米公司 Vela 研发团队
- 项目难度：中高
- 项目链接：
  `https://github.com/oscomp/proj182-xiaomi-nuttx-smp-scheduler`

题目基本情况

项目要求：

基于 NuttX 已有的多核支持，实现一个符合 POSIX 规范的 SMP 多核调度器。

1. 符合 POSIX 规范，支持 FIFO、RR、Sporadic 调度策略
2. 完成每个 cpu core local 的调度器，减少当前代码中全局锁的使用
3. 对称式调度器（每个核心维护自己的 task list）
4. 适配 K210 板卡，并考虑多平台的可扩展（ARM、Xtensa）

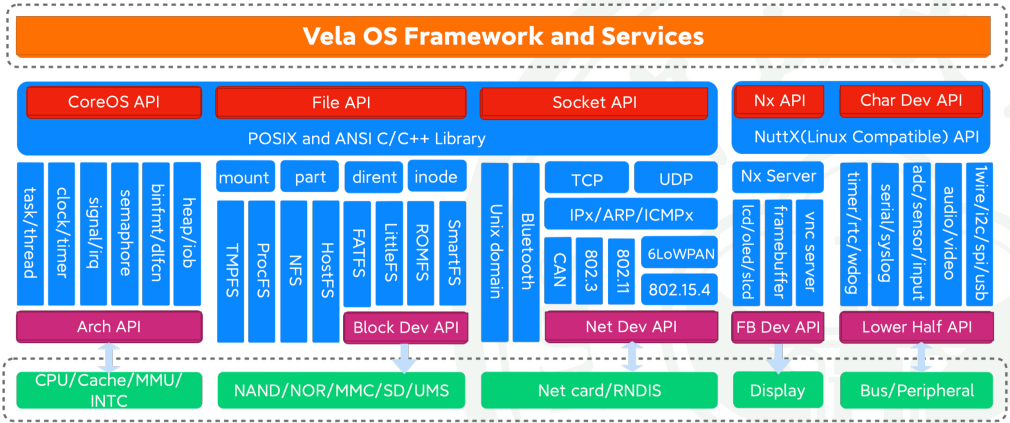
目录

Apache NuttX 简介

- Apache NuttX 是一个开源的实时操作系统；
- 支持 8 至 64 位的多种架构处理器，同时提供 SMP 和 AMP 的多核支持；
- 接口完全兼容 POSIX 和 ANSI 标准；
- 子系统之间通过 kconfig 控制，可以按照需要进行裁剪 [1]；
- 在 Apache 基金会下开源，遵循 Apache-2.0 license 开源协议。

Apache NuttX 简介

NuttX RTOS 的系统结构概述 [2]:



NuttX SMP 支持情况

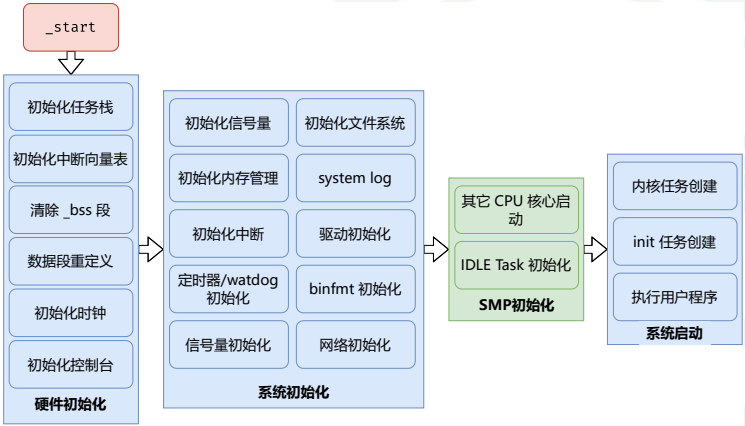
NuttX 提供对于 SMP 的支持，通过宏定义进行 SMP 相关功能的开关和配置。

- `CONFIG_SMP`：在多 CPU 平台上启用对称多处理 (SMP) 支持。
- `CONFIG_SMP_NCPUS`：标识将用于 SMP 的处理器支持的 CPU 数量。
- `CONFIG_SMP_IDLETHREAD_STACKSIZE`：提供 CPUS 1 至 (`CONFIG_SMP_NCPUS`-1) 上 IDLE 任务的堆栈大小。

同时 NuttX 还在相关子系统的实现中为 SMP 支持提供了一系列的支持 [3]。

NuttX SMP 下的系统启动流程

NuttX 多核系统在系统启动前期，引导程序只会引导一个 CPU 进行启动和最初的初始化，在最初的操作系统初始化完毕，正常多任务处理之前，其他的 CPU 才会被启动 [4]。



信号量、锁和临界区

- NuttX 中断处理不能嵌套，有中断上下半部、Work Queue 等的支持；
- 锁的实现主要也是自旋锁和忙等待；
- 多核情况下的临界区实现更为复杂：
  - 单核情况下禁用中断就可以禁止上下文切换，实现临界区访问控制；
  - SMP 情况下一把大锁解决不了问题。
    1. 使用 `up_irq_save()` 函数来通过进程间的通信来强制禁用所有核心上的中断
    2. 给 `g_cpu_irqset` 上锁并立即刷新缓存
    3. 在调用临界区的程序 TCB 上留下对应的标志以防止被抢占导致死锁。
  - 特殊情况：ARM 架构下的 GIC 支持：SGI 无法被禁用。[3]

多核调度的数据结构支持

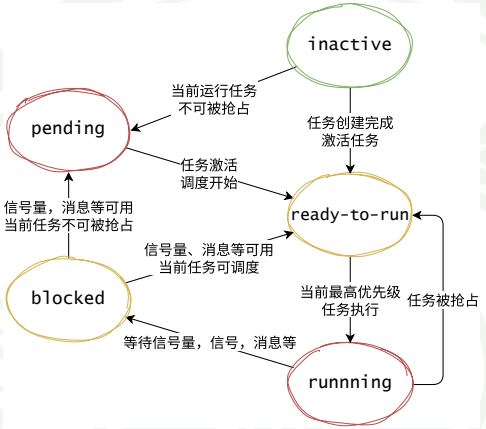
```
struct tasklist_s
{
  DSEG dq_queue_t *list;
  uint8_t attr;
};

extern dq_queue_t g_readytorun;
#ifdef CONFIG_SMP
extern dq_queue_t
↪  g_assignedtasks[CONFIG_SMP_NCPUS];
#endif
```

```
#ifdef CONFIG_SMP
extern volatile spinlock_t
↪  g_cpu_schedlock;
extern volatile spinlock_t
↪  g_cpu_locksetlock;
extern volatile cpu_set_t
↪  g_cpu_lockset;
extern volatile spinlock_t
↪  g_cpu_tasklistlock;
#endif /* CONFIG_SMP */
```

NuttX 任务、线程与调度粒度

- NuttX 不支持**进程**，由于其不要求硬件强制具有 MMU;
- **任务**（Task）是内存分配的基本单位，在 RTOS 中相当于进程;
- NuttX 也支持 POSIX pthreads;
- 对于 NuttX 来说，进程和线程都维护一个 TCB 表，其中的 pid 字段唯一标识线程/进程 [5];
- TCB 是 NuttX 调度的**基本单位**。



NuttX 调度方式与调度时机

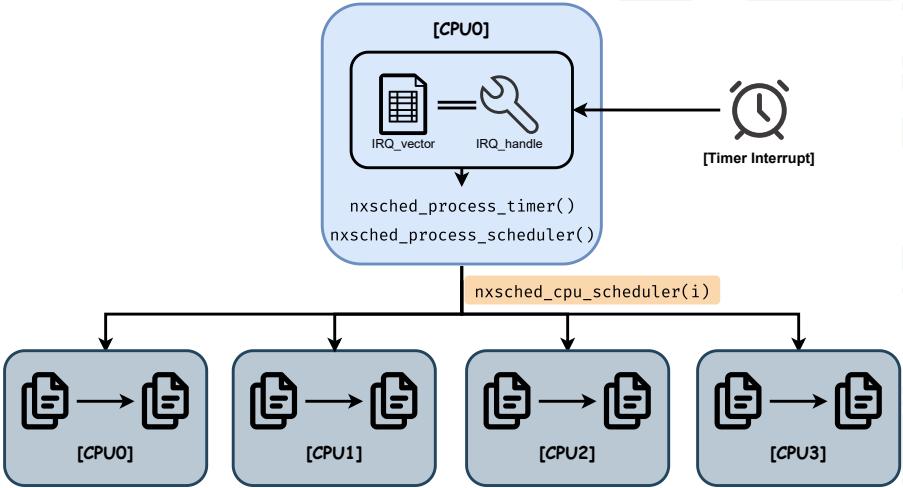
Apache NuttX 唤醒调度器的入口：

- 任务创建时产生的调度；
- 定时器中断产生的调度 （处理重点）
- sleep/usleep/nanosleep 产生的调度；
- 调用互斥锁导致的调度；
- 资源不可用导致的调度；
- ……

支持的任务/线程的调度方式：

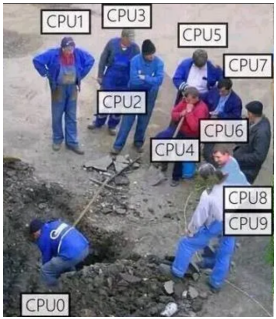
- FIFO：RTOS 的默认调度方式；
- Round-Robin ： 通过宏 `CONFIG_RR_INTERVAL` 控制；
- Sporadic: 通过宏 `CONFIG_SCHED_SPORADIC` 控制。

定时器中断产生的调度


[CPU0]
IRQ_vector
IRQ_handle
[Timer Interrupt]
nxsched_process_timer()
nxsched_process_scheduler()
nxsched_cpu_scheduler(i)
[CPU0]
[CPU1]
[CPU2]
[CPU3]

现有实现中的问题

定时器中断产生的调度过程中，由主核心 [CPU 0] 在临界区中完成所有核心的调度工作。




```
flags = enter_critical_section();
/* Perform scheduler operations on all CPUs */
for (i = 0; i < CONFIG_SMP_NCPUS; i++)
  {
    nxsched_cpu_scheduler(i);
  }
leave_critical_section(flags);
```

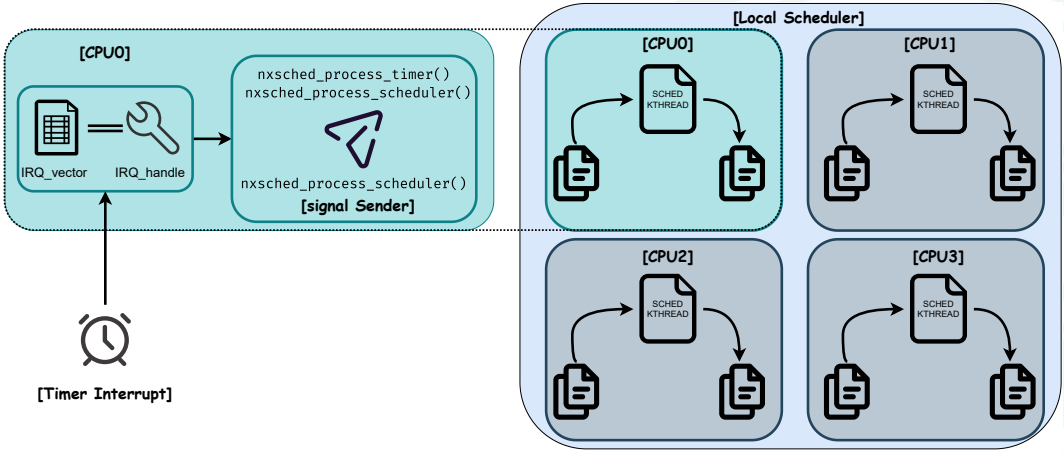
Tickless

- NuttX 原生提供 Tickless 支持
- 通过宏 CONFIG_SCHED_TICKLESS 来控制是否开启对应支持；
- 在相关调度实现文件中，我们发现了熟悉的实现逻辑：

```
for (i = 0; i < CONFIG_SMP_NCPUS; i++)
{
  timeslice = nxsched_cpu_scheduler
              (i, ticks, noswitches);
  if (timeslice > 0 && timeslice < minslice)
    {
      minslice = timeslice;
    }
}
```

目录

调度整体设计思路


[CPU0]
IRQ_vector
IRQ_handle
nxsched_process_timer()
nxsched_process_scheduler()
nxsched_process_scheduler()
[signal Sender]
[Timer Interrupt]
[Local Scheduler]
[CPU0]
SCHED KTHREAD
[CPU1]
SCHED KTHREAD
[CPU2]
SCHED KTHREAD
[CPU3]
SCHED KTHREAD

调度系统初始化

由于我们设计的是一个两段式的调度器，初始化也分为两段，分别是调度系统初始化和服务注册。

- 调度系统初始化阶段：
  - 核心调度函数注册；
  - 任务数据结构的初始化；
  - 调度数据结构初始化；
  - 全局调度锁的初始化
  - ...
- 在其他子系统初始化之前，使用系统原有的初始化函数。

- 服务注册阶段：
  - 调度线程的创建和核心绑定
  - 每 CPU 本地调度器的初始化；
  - 调度信号量的初始化；
  - 每 CPU 调度锁的初始化
- 在相关子系统都初始化完成之后，项目所添加的内容。

调度系统初始化

- 实现过程中的问题：NuttX 内核线程（kthread）创建接口中没有提供核心绑定相关的接口。
- 解决方案：在 task_create.c 中创建新的函数手动初始化 TCB 数据结构。

```
int kthread_create_with_cpu(FAR const char *name,
        int priority, int stack_size, u_int16_t cpu,
        main_t entry, FAR char * const argv[]);
```

- 其中有：

```
tcb->cmn.flags = (TCB_FLAG_TTYPE_KERNEL   |
                  TCB_FLAG_NONCANCELABLE |
                  TCB_FLAG_CPU_LOCKED     );
tcb->cmn.cpu = cpu;
```

时钟中断发生之后

- 在调研阶段我们已经梳理了 NuttX 时钟中断处理流程。
- 调度流程第一段：CPU0 收到时钟中断之后
  1. 发送信号给所有核心本地上的调度线程；
  2. 将调度线程调度到 CPU 前台。
- 解决方案：使用信号量函数：

```
int nxsem_post(FAR sem_t *sem)
```

时钟中断发生之后

```
flags = enter_critical_section();
/* Perform scheduler operations on
↪  all CPUs */
for (i = 0; i < CONFIG_SMP_NCPUS;
↪  i++)
  {
    nxsched_cpu_scheduler(i);
  }
leave_critical_section(flags);
```

改进前的 for 循环实现

```
for (i = 0; i < CONFIG_SMP_NCPUS; i++)
  {
    if(g_cpu_scheduler_sem[i].semcount
↪  <= 0)
    {
      sem_post(&g_cpu_scheduler_sem[i]);
    }
  }
```

改进后的 for 循环实现

调度系统初始化

为什么 sem_post() 可以保证第一段调度的两个要求？

```
if (sem_count <= 0)
{
  stcb = (FAR struct tcb_s
↪  *)dq_remfirst(SEM_WAITLIST(sem));
  if (stcb != NULL)
    {
      FAR struct tcb_s *rtcb =
↪  this_task();
      nxsem_add_holder_tcb(stcb, sem);
```

```
      // Something about watchdog

      if (nxsched_add_readytorun(stcb))
        {
          up_switch_context(stcb, rtcb);
        }
    }
#endif
}
```

关键在于执行一次上下文切换！

核心上的队列调度

- 调度流程第二段：本地核心上的调度。
- 现有一个可以在核心上完成调度的函数：

```
static inline void nxsched_cpu_scheduler(int cpu)
```

- 我们自然复用了这个函数，只需要为这个函数完成一个和本地调度线程绑定的一个包装函数：

```
static int nxsched_cpu_scheduler_inpoint(int argc, const char** argv)
```

核心上的队列调度

核心调度策略：

```
for (;;){
  ret = nxsem_wait_uninterruptible(&g_cpu_scheduler_sem[cpu]);
  if (ret < 0) {
      DEBUGASSERT(ret == -EINTR);
      continue;
    }
  nxsched_cpu_scheduler(cpu, ticks, noswitches);
}
```

实验环境

模拟环境：

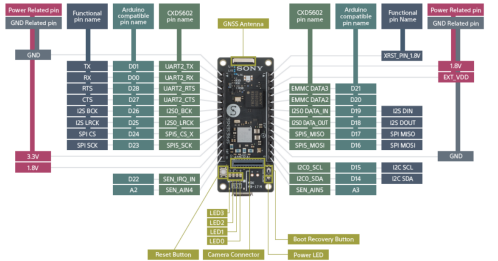
- 借助 Docker 和 qemu 我们构造了用于代码编写，编译和调试的虚拟环境。
- 根据要求和队伍和小米 NuttX 团队的沟通结果，我们构建了多个架构的虚拟环境，主要使用 Armv8 和 RISC-V 64 两个架构的虚拟环境，其对应的 Dockerfile 和配置文档均开源在在队伍仓库中。
- 借助 VSCode 的 Dev Container 我们可以一键式地完成 NuttX 的环境配置，而借助 GitHub 提供的 Dev Container 功能我们更是可以在浏览器中就可以完成项目的开发，编译和调试。

实验环境

实机环境：

- 我们使用一块 Sony Spresense B1 进行实机环境的烧写和测试。

类型	配置详情
型号	CXD5602PWBMAIN1
尺寸	50.0mm x 20.6mm
CPU	ARM® Cortex®-M4F x 6 核
最大时钟频率	156MHz
SRAM	1.5MB
flash memory	8MB



实验验证

- 使用 NuttX 仓库中自有的 SMP 测试用例作为我们调度的测试用例；
  - 定义在 nuttx-app 仓库下的 `apps/testing/smp/` 文件夹中；
  - 在编译时只需要在 Kconfig 文件中国中加入对应的宏定义 `CONFIG_TESTING_SMP`，在运行时就可以在 NuttX Shell 使用 `smp` 命令中调用对应的测试应用程序了；
- 在我们配置的虚拟环境和实机环境中都进行测试脚本的运行：
  - 可以发现我们的调度器可以正常进行调度，运行结果和原有调度器一致。
  - 加入对应的时钟探针函数，发现此时调度其实并没有显著性的差别，这将会是未来的一个优化方向。

实验结果

```
nsh: mkfatfs: command not found

NuttShell (NSH) NuttX-12.0.0
nsh> [CPU0] nx_start: CPU0: Beginning Idle Loop
smp
[CPU0] task_spawn: name=smp entry=0x402a106c file_actions=0x402f40
90 attr=0x402f4098 argv=0x402f41e0
[CPU0] spawn_execattrs: Setting policy=2 priority=100 for pid=10
  Main[0]: Running on CPU0
  Main[0]: Initializing barrier
  Main[0]: Thread 1 created
Thread[1]: Started
Thread[1]: Running on CPU1
  Main[0]: Thread 2 created
Thread[2]: Started
Thread[2]: Running on CPU2
  Main[0]: Thread 3 created
Thread[3]: Started
Thread[3]: Running on CPU3
  Main[0]: Thread 4 created
  Main[0]: Thread 5 created
  Main[0]: Thread 6 created
  Main[0]: Thread 7 created
  Main[0]: Thread 8 created
[CPU0] pthread_join: thread=11 group=0x402e47a0
[CPU0] pthread_join: Thread is still running
Thread[4]: Started
Thread[4]: Running on CPU0
Thread[5]: Started
Thread[5]: Running on CPU1
Thread[6]: Started
Thread[6]: Running on CPU2
Thread[7]: Started
Thread[7]: Running on CPU3
Thread[8]: Started
Thread[8]: Running on CPU0
Thread[2]: Now running on CPU3
Thread[3]: Now running on CPU2
Thread[4]: Now running on CPU2
Thread[5]: Now running on CPU2
Thread[7]: Now running on CPU2
```

```
d[3]: Done
[CPU3] pthread_completejoin: pThreaid=20 exit_d[valu8]:e=0 gr Now
ourunng=ing on0x4 CPU1
02e92e0
U3P[CPU1] nx_pthread_exit: exit_value=0
] pthread[CPU1] pthread_complete_nojointi: fywpiaid=te22 exit_valu
e=0 group=0x402e92e0
rs: pjoin=0x403885c0
Thread[7]: Now running on CPU1
[CPU1] pthread_notifywaiters:[CPU pjoi3] nx_pthreadn=0x_e4039090
xit: exit_value=0
[CPU3] pthread_comThpletejoin: pread[7]: Did=1one
3 exit_value=0 group=0x402e92e0
[CPU3] pthread_notifywaiters: pjoiThread[1]:n=0x402e68c0
 Now running on CPU0
[CPU1] nx_pthread_exit: exit_value=0
Thread[1]: Done
[CPU1] [CpthreadPU0] nx_pthread_exit: exit_value=0
_completejoin: pid=T21 exit_vhread[alue5]=0: Done
 group[CPU0] pt=[CPUhre20a] nx_pdxthread_ex402e92it: ee0
x[CPUit_value=0
_complet[CPU2] ptejo1] pthreahread_in:cd pid=omp_notifyletejoinw1:
 1aiters: pjo exit_value=0 group=0x40in=0x4pi038c760
2e92e0d=15 exit
_val[CPU0] pthuere=0 ad_notifywaiters: pjoin=0x402e2group4a0
=0x402e92e0
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_notifywaiters: pjoin=0x4038420
[CPU0] pthread_destroyjoin: pjoin=0x402e24a0
[CPU0] pthread_join: Returning 0
  Main[0]: Thread 1 completed with result=0
[CPU0] pthread_join: thread=12 group=0x402e92e0
[CPU0] pthread_join: Thread has terminated
[CPU0] pthread_join: exit_value=0
[CPU0] pthread_destroyjoin: pjoin=0x402e96e0
[CPU0] pthread_join: Returning 0
  Main[0]: Thread 2 completed with result=0
[CPU0] pthread_join: thread=13 group=0x402e92e0
[CPU0] pthread_join: Thread has terminated
[CPU0] pthread_join: exit_value=0
[CPU0] pthread_destroyjoin: pjoin=0x402e68c0
```

```
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_destroyjoin: pjoin=0x402e9700
[CPU1] pthread_join: Returning 0
  Main[0]: Thread 2 completed with result=0
[CPU1] pthread_join: thread=13 group=0x402e92e0
[CPU1] pthread_join: Thread has terminated
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_destroyjoin: pjoin=0x402e68c0
[CPU1] pthread_join: Returning 0
  Main[0]: Thread 3 completed with result=0
[CPU1] pthread_join: thread=14 group=0x402e92e0
[CPU1] pthread_join: Thread has terminated
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_destroyjoin: pjoin=0x402e6a40
[CPU1] pthread_join: Returning 0
  Main[0]: Thread 4 completed with result=0
[CPU1] pthread_join: thread=15 group=0x402e92e0
[CPU1] pthread_join: Thread has terminated
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_destroyjoin: pjoin=0x402e6bc0
[CPU1] pthread_join: Returning 0
  Main[0]: Thread 5 completed with result=0
[CPU1] pthread_join: thread=20 group=0x402e92e0
[CPU1] pthread_join: Thread has terminated
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_destroyjoin: pjoin=0x4038560
[CPU1] pthread_join: Returning 0
  Main[0]: Thread 6 completed with result=0
[CPU1] pthread_join: thread=21 group=0x402e92e0
[CPU1] pthread_join: Thread has terminated
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_destroyjoin: pjoin=0x4038c700
[CPU1] pthread_join: Returning 0
  Main[0]: Thread 7 completed with result=0
[CPU1] pthread_join: thread=22 group=0x402e92e0
[CPU1] pthread_join: Thread has terminated
[CPU1] pthread_join: exit_value=0
[CPU1] pthread_destroyjoin: pjoin=0x403908a0
[CPU1] pthread_join: Returning 0
  Main[0]: Thread 8 completed with result=0
nsh>
```

目录

项目实现功能

项目基本实现题目要求的预期目标。

根据题目要求，该项目的预期目标是：

1. 符合 POSIX 规范，支持 FIFO、RR、Sporadic 调度策略
2. 对称式调度器（每个核心维护自己的 task list）or 其他更优的无锁实现
3. 适配 K210 板卡，并考虑多平台的可扩展（ARM、Xtensa）

对照我们的实现：

1. 改动并没有改变 NuttX 向上的接口，仍然符合 POSIX 规范；同时也并没有改变操作系统调度器的适配性；
2. 实现了 CPU-local 的本地调度支持，构建了题目中要求的对称式调度器；
3. 并没有改变驱动和架构相关的代码接口部分，在理论上不会对 NuttX 的硬件适配性造成干扰。

不足之处和未来改进方向

开源项目最终的目标还是要回馈开源社区，面向更高层次的目标，我们的项目还有一定的不足，这也是我们未来努力的方向。

1. 代码接口不够规范，并没有按照社区要求进行代码编写和格式化；
2. 文档不够完善，缺少完整的接口描述文档，接口暴露有不合理的地方；
3. 测试不够完全，缺少对于调度正确性的完整单元测试和集成测试；
4. 在性能上仍有不小的改进空间，未来考虑改进本地调度器实现方式；
5. 由于 NuttX 硬件适配的问题，Arm 下有偶发性死锁问题存在。

参考文献

[1] D. Sánchez-López, "Low power embedded software optimization for the nuttx rtos," 2013.

[2] M. Lenc, "Open rapid control prototyping and real-time systems," 2022.

[3] "Smp - nuttx - apache software foundation."

[4] D. J. Barker and D. C. Stuckey, "A review of soluble microbial products (smp) in wastewater treatment systems," *Water research*, vol. 33, no. 14, pp. 3063–3082, 1999.

[5] "Architecture apis —nuttx latest documentation."

目录

Thanks Q&A

感谢各位评委老师的垂听

请提出宝贵意见和问题

基于 NuttX RTOS 的符合 POSIX 规范和 SMP 系统的高效任务调度器实现

202310558111463 铃芽小姐来死锁

杨翼飞[2] 陈冠一[1] 周桐欣[1] 赵帅[1,*] 黎卫兵[1,*]

[1] 中山大学 计算机学院（软件学院）
[2] 中山大学 软件工程学院
[*] 指导教师

中山大學
SUN YAT-SEN UNIVERSITY