SpaceChain OS 实时性

测评报告

2018/05/12

修订历史		
日期	版本号	变更内容
2018 年 05 月 12 日	V1.0	新建文档

目录

1. 测试目的

本测试报告为 SpaceChain OS 实时性测试报告；本报告目的在于总结测试阶段的测试以及对测试结果进行分析。

2. 测试设计

2.1 测试操作系统简介

2.1.1 SpaceChain OS 操作系统

SpaceChain OS 源自于 SylixOS。 SylixOS 是一款嵌入式硬实时操作系统，同其类似的操作系统，全球比较知名的还有 VxWorks(主要应用于航空航天、军事与工业自动化领域)、RTEMS(起源于美国国防部导弹与火箭控制实时系统)、ThreadX(主要应用于航空航天与数码通讯)等。 从全球范围上看，SpaceChain OS 作为实时操作系统的后来者，在设计思路上借鉴了众多实时操作系统的设计思想，其中就包括 RTEMS、VxWorks、ThreadX 等，使得具体性能参数上达到或超过了众多实时操作系统的水平，成为国内实时操作系统的最优秀代表之一。

2.1.2 Linux 操作系统

Linux 是一套免费使用和自由传播的类 Unix 操作系统，它是一个符合 POSIX 标准的多用户、多任务操作系统。它支持 32 位和 64 位以及多核心 CPU。

2.1.3 Linux+RT 补丁

标准的 Linux 内核只能满足软实时应用的需求，但不对截止的时间点作保证。Linux 实时抢占补丁 PREEMPT_RT 使 Linux 增加了硬实时能力。

Linux 实时抢占补丁引起了工业界的关注；由于它简洁的设计和与内核 mainline 的一致性，使得它出现在专业的视频到工业控制领域。

2.2 测试工具简介

2.2.1 Cyclictest

在本次的测试过程中，我们使用 Cyclictest 工具进行实时性测试，Cyclictest 是 rt-tests 项目下使用最广泛的一个测试工具。它是一个高精度的测试程序，一般用于测试操作系统的延迟，从而判断操作系统的实时性。

下面结合源代码来分析 Cyclictest 的原理。

rt-tests/src/cyclictest/cyclictest.c 的 main 函数如程序清单 2.1 所示。

程序清单 2.1 cyclictest main 函数

```
int main(int argc, char **argv)
{       ...
        stat->min = 1000000;
        stat->max = 0;
        stat->avg = 0.0;
        stat->threadstarted = 1;
        status = pthread_create(&stat->thread, &attr, timerthread, par);
         ...
}
```

main 函数中只是初始化了相关的变量，而具体测试工作是通过创建的线程来完成的，在测试线程中进行测试和记录，测试线程如程序清单 2.2 所示。

程序清单 2.2 cyclictest 测试线程

```
void *timerthread(void *param)
{       ...
        interval.tv_sec = par->interval                         // 将参数中的间隔数赋给函数中的间隔数（interval）
        interval.tv_nsec = (par->interval % USEC_PER_SEC) * 1000;
         ...
        /* Get current time    */
        clock_gettime(par->clock, &now);                        // 获取当前时间，存在 now 中

        next = now;                                             // 这三行是将当前时间（now 的值）
        next.tv_sec += interval.tv_sec;                         // 加上间隔数（interval）
        next.tv_nsec += interval.tv_nsec;                       // 算出下次间隔的时间，存在 next 中
        tsnorm(&next);
         ...
         /* Wait for next period */                             // 等到下次循环
                ...
         if ((ret = clock_gettime(par->clock, &now))) {         // 记录下次循环的时间到 now 中，此时 now 值中存的数是真实的下次循环的值，而上面存在 next 的值是上次循环加上间隔值，所以是理论上的下个循环的值
                    if (ret != EINTR)
                        warn("clock_getttime() failed. errno: %d\n", errno);
                    goto out;
                }
         if (use_nsecs)
```

```
        diff = calcdiff_ns(now, next);        // 上面已经说明，now 中是下次循环的真值，而 next 是理论的值，所以两者的差就是延时，延时赋值给 diff
    else
        diff = calcdiff(now, next);
    if (diff < stat->min)                     // 假如延时比 min 小，将 min 改为这个更小的延时值 diff
        stat->min = diff;
    if (diff > stat->max) {                   // 假如延时比 max 大，将 max 改为这个更大的延时值 diff
        stat->max = diff;
        if (refresh_on_max)
            pthread_cond_signal(&refresh_on_max_cond);
    }
    stat->avg += (double) diff;               // 计算新的平均延时
     ...
    /* Update the histogram */                // 更新 histogram 中存的延时统计数据
    if (histogram) {
     if (diff >= histogram) {                 // 假如延时比 histogram 大，添加一次溢出
            stat->hist_overflow++;
                                    if (stat->num_outliers < histogram)
                stat->outliers[stat->num_outliers++] = stat->cycles;
        }
        else                                  // 如果没有溢出，将 histogram 中的相应值加 1
            stat->hist_array[diff]++;
    }

    stat->cycles++;                           // 循环加 1

    next.tv_sec += interval.tv_sec;           // 继续计算下次循环的值 ...
    next.tv_nsec += interval.tv_nsec;
     ...
}
```

Cyclictest 通过测试线程进行循环调度，设定调度的间隔时间，比较真实时间与计算时间的差值来获得实际过程中的延迟时间，通过延迟时间的大小来确定操作系统的实时性。

2.2.2 Hackbench

Hackbench 是一个测试 Linux 进程调度器的性能、开销和伸缩性的基准测试程序。Hackbench 测试可以运行在不同的硬件平台和不同的 Linux 操作系统版本上，通过模拟一组 C/S 模式的客户端进程和服务器进程间的通信来测试 Linux 进程调度器的性能、开销和伸缩

性。

Hackbench 测试分为若干组进行，组数 num_group 由命令行给出，每个组又有 num_fds 组读写进程，num_fds 的初值设置为 20。在每一个测试组中，创建 num_fds 个客户端进程（读进程）和 num_fds 个服务器进程（写进程）分别负责信息的读写工作，创建 num_fds 个管道用于读写进程间的通信。写进程负责向管道中写数据，读进程负责从管道中读取数据。

通过它可以不断地进行进程间通信以及进程创建，从而提供一个压力环境给 Cyclictest 测试程序。

2.3 测试环境与配置

2.3.1 测试环境

测试环境如表 2.1 所示。

表 2.1 测试环境

软硬件	型号
单核测试开发板	OK335xS
多核测试开发板	E9
串口输出客户端	PuttY

2.3.2 相关测试操作系统配置

相关测试操作系统配置如表 2.2 所示。

表 2.2 相关测试操作系统配置

操作系统	版本
SpaceChain OS 操作系统	STABLE-1.1.1
OK335xS 配套 Linux 操作系统	3.2.0
Linux 操作系统配套 RT 补丁	3.2.0-rt10
E9 配套 Linux 操作系统	Linux EmbedSky 3.0.35

2.4 测试方法

2.4.1 单核测试

在单核测试中，我们使用 OK335xS 开发板进行测试，该开发板有配套的 Linux 系统，内核版本号为 3.2.0。以此为基础进行实时性测试，目的是评测 SpaceChain OS 和 Linux 及 Linux+RT 之间的实时性差异，主要的测试有以下两方面：

1. 无压力测试：

分别在 Linux、Linux+RT 以及 SpaceChain OS 中进行 1000000 次循环的 cyclictest 测试，

每次循环的间隔为 10ms。

2. 有压力测试：

分别在 Linux、Linux+RT 以及 SpaceChain OS 中进行 100000 次循环的 cyclictest 的测试，每次循环的间隔为 10ms，在测试的同时，运行 hackbench 提供压力环境，从而完成有压力测试。

2.4.2 多核测试

在多核测试中，我们使用 E9 开发板进行测试，该开发板有配套的 Linux 系统，内核版本号为 3.0.35。

测试方法依然使用 cyclictest 和 hackbench 进行测试。

因为该板配套的 Linux 打上 RT 补丁后无法正常启动，所以该测试只在 SpaceChain OS 和其配套的 Linux 系统中进行，同样分别进行无压力和有压力测试，并且与 SpaceChain OS 单核中的实时性进行比较。

3. 测试情况

3.1 测试执行情况

3.1.1 测试范围和要求

测试范围包括：单核有压力测试、无压力测试、多核无压力测试、多核有压力测试。

3.1.2 测试时间

测试时间如表 3.1 所示。

表 3.1 测试时间

测试类型	测试时间
单核 Linux 无压力测试	2015 年 6 月 30 日 9 点-12 点
单核 SpaceChain OS 无压力测试	2015 年 6 月 30 日 13 点-16 点
单核 Linux+RT 无压力测试	2015 年 6 月 30 日 17 点-20 点
单核 Linux 有压力测试	2015 年 6 月 30 日 20 点-21 点
单核 SpaceChain OS 有压力测试	2015 年 6 月 30 日 21 点-22 点
单核 Linux+RT 有压力测试	2015 年 6 月 30 日 22 点-23 点
多核 SpaceChain OS、Linux 测试	2015 年 7 月 1 日 9 点-10 点

3.1.3 测试机构和人员

本次测试由清华大学陈一璋博士完成。

3.2 测试结果统计

3.2.1 单核无压力结果统计

- SpaceChain OS：

单核 100w 次无压力测试结果如图 3.1 所示。

```
T: 0 (67174458) P: 2 I:10000 C:1000000 Min:      3 Act:    4 Avg:    4 Max:      35
```

图 3.1 SpaceChain OS 单核 100w 次无压力测试结果

测试结果汇总如下表：

最大延迟时间	35us
最小延迟时间	3us
平均延迟时间	4us

- Linux：

单核 100w 次无压力测试结果如图 3.2 所示。

```
# Total: 001000000
# Min Latencies: 00007
# Avg Latencies: 00013
# Max Latencies: 00717
```

图 3.2 Linux 单核 100w 次无压力测试结果

测试结果汇总如下表：

最大延迟时间	717us
最小延迟时间	7us
平均延迟时间	13us

- Linux+RT 补丁：

单核 100w 次无压力测试结果如图 3.3 所示。

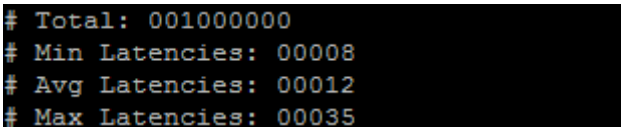

图 3.3 Linux+RT 补丁单核 100w 次无压力测试结果

测试结果汇总如下表：

最大延迟时间	35us
最小延迟时间	8us
平均延迟时间	12us

3.2.2 单核有压力结果统计

- SpaceChain OS：

单核 10w 次有压力测试结果如图 3.4 所示。

图 3.4 SpaceChain OS 单核 10w 次有压力测试结果

测试结果汇总如下表：

最大延迟时间	62us
最小延迟时间	3us
平均延迟时间	4us

- Linux:

单核 10w 次有压力测试结果如图 3.5 所示。

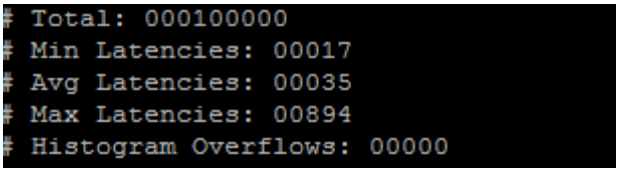

图 3.5 Linux 单核 10w 次有压力测试结果

测试结果汇总如下表：

最大延迟时间	894us
最小延迟时间	17us
平均延迟时间	35us

- Linux+RT :

单核 10w 次有压力测试结果如图 3.6 所示。

图 3.6 Linux+RT 补丁单核 10 万次有压力测试结果

测试结果汇总如下表：

最大延迟时间	67us
最小延迟时间	8us
平均延迟时间	31us

3.2.3 多核无压力结果统计

- SpaceChain OS：

多核 10w 次无压力测试结果如图 3.7 所示。

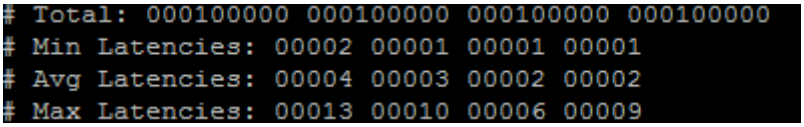

图 3.7 SpaceChain OS 多核 10w 次无压力测试结果

测试结果汇总如下表：

	CPU#0	CPU#1	CPU#2	CPU#3
最小的延迟数	2us	1us	1us	1us
平均的延迟数	4us	3us	2us	2us
最大的延迟数	13us	10us	6us	9us

- Linux：

多核 10w 次无压力测试结果如图 3.8 所示。

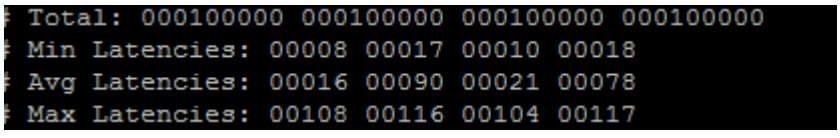

图 3.8 Linux 多核 10w 次无压力测试结果

测试结果汇总如下表：

	CPU#0	CPU#1	CPU#2	CPU#3
最小的延迟数	8us	17us	10us	18us
平均的延迟数	16us	90us	21us	78us

最大的延迟数	108us	116us	104us	117us

3.2.4 多核有压力测试结果统计

◆ SpaceChain OS：

多核 10w 次有压力测试结果如图 3.9 所示。

```
# Total: 000100000 000100000 000100000 000100000
# Min Latencies: 00002 00002 00002 00002
# Avg Latencies: 00005 00005 00005 00005
# Max Latencies: 00017 00019 00016 00025
```

图 3.9 SpaceChain OS 多核 10w 次有压力测试结果

测试结果汇总如下表：

	CPU#0	CPU#1	CPU#2	CPU#3
最小的延迟数	2us	2us	2us	2us
平均的延迟数	5us	5us	5us	5us
最大的延迟数	17us	19us	16us	25us

◆ Linux：

多核 10w 次有压力测试结果如图 3.10 所示。

```
# Total: 000100000 000100000 000100000 000100000
# Min Latencies: 00009 00017 00013 00018
# Avg Latencies: 00016 00076 00020 00063
# Max Latencies: 00102 00121 00106 00093
```

图 3.10 Linux 多核 10w 次无压力测试结果

测试结果汇总如下表：

	CPU#0	CPU#1	CPU#2	CPU#3
最小的延迟数	9us	17us	13us	18us
平均的延迟数	16us	76us	20us	63us
最大的延迟数	102us	121us	106us	93us

3.3 测试结果分析

3.3.1 单核无压力结果分析

首先从统计结果可以得出没有实时补丁的 Linux 操作系统实时性最差，最大延迟高达717us，所以在接下来的分析以 SpaceChain OS 与 Linux+RT 为主，以下是通过使用 cyclictest 工具记录 100w 次无压力延迟数所画的统计图，如图 3.11 所示。

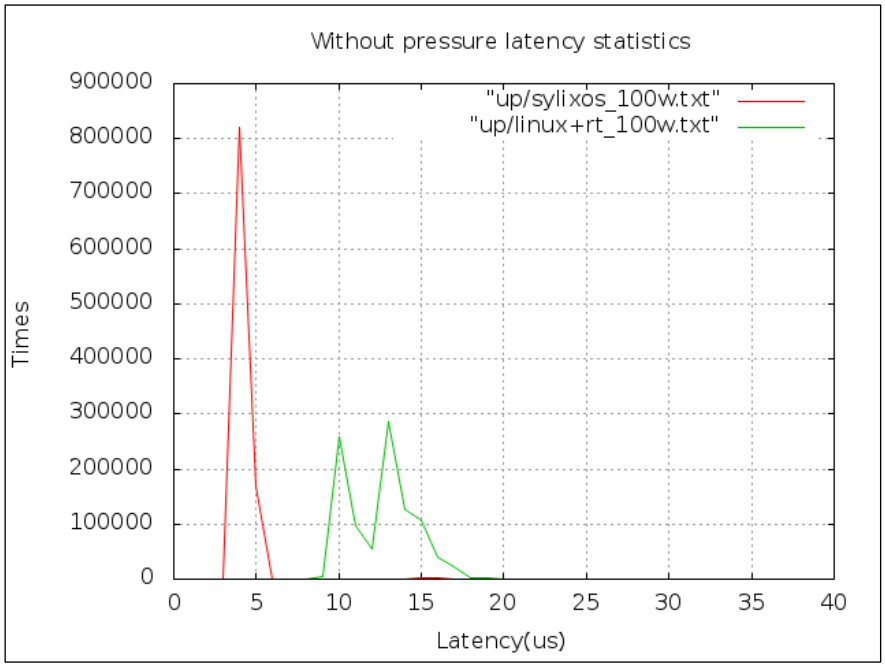

图 3.11 SpaceChain OS、Linux+RT 单核 100w 无压力次延迟次数统计

通过统计图可以看出，在单核无压力 100w 次测试中，SpaceChain OS 与 Linux+RT 的最

大延迟时间相同，均为 35us；但是 SpaceChain OS 的平均延迟时间较为低，大部分集中在 4us，而 Linux+RT 则 13us 的延迟数最多。

比较结果汇总如下表：

	SpaceChain OS	Linux+RT	Linux	比较结果
最大延迟时间（us）	35	35	717	SpaceChain OS=Linux+RT<linux
最小延迟时间（us）	3	8	7	SpaceChain OS<linux<Linux+RT
平均延迟时间（us）	4	12	13	SpaceChain OS<Linux+RT<linux

3.3.2 单核有压力结果分析

首先从统计结果可以得出没有实时补丁的 Linux 操作系统实时性最差，最大延迟高达 894us，所以在接下来的分析以 SpaceChain OS 与 Linux+RT 为主，以下是通过使用 cyclictest 工具记录 10w 次有压力延迟数所画的统计图，如图 3.12 所示。

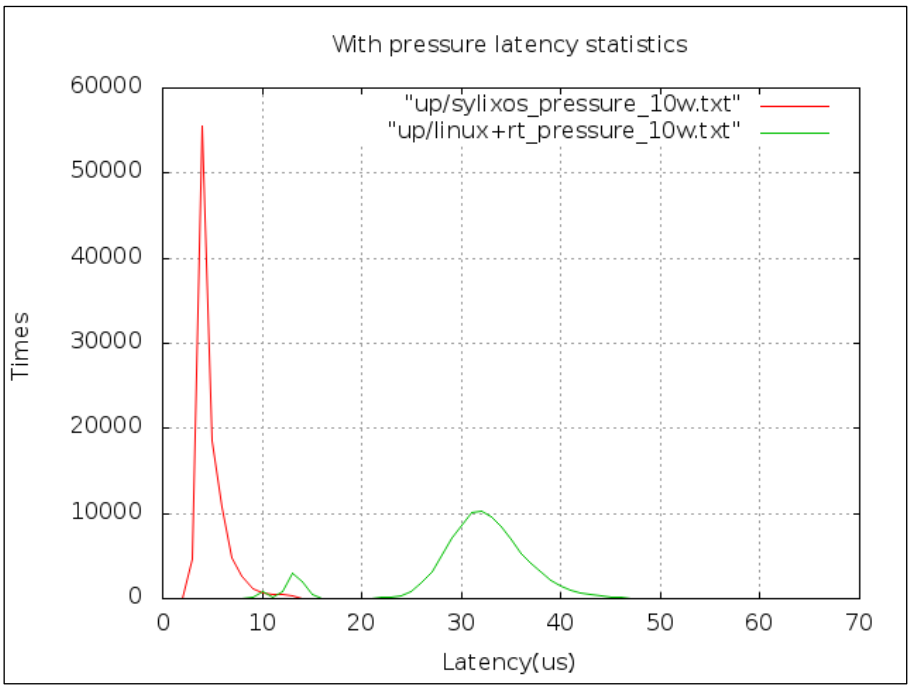

图 3.12　SpaceChain OS、Linux+RT 单核 10w 有压力延迟次数统计

通过统计图可以看出，在单核有压力 10w 次测试中，SpaceChain OS 与 Linux+RT 的最大延迟数都较没有压力的情况下升高，SpaceChain OS 的最大延迟时间是 62us，而 Linux+RT 中的最大延迟时间为 67us，SpaceChain OS 的最大延迟时间较低，并且通过统计图中我们可以看出在压力的情况下，SpaceChain OS 的平均延迟时间较低。

比较结果汇总如下表：

	SpaceChain OS	Linux+RT	Linux	比较结果
最大延迟时间（us）	62	67	894	SpaceChain OS < Linux+RT < Linux
最小延迟时间（us）	2	8	17	SpaceChain OS < Linux+RT < Linux
平均延迟时间（us）	4	31	35	SpaceChain OS < Linux+RT < Linux

3.3.3 多核无压力结果分析

比较结果汇总如下表：

	SpaceChain OS	Linux	比较结果
最大延迟时间（us）	13	117	SpaceChain OS << Linux
最小延迟时间（us）	2	18	SpaceChain OS < Linux
平均延迟时间（us）	4	90	SpaceChain OS < Linux

从统计结果可以看出，在无压力的情况下 SpaceChain OS 的最大延迟时间明显小于 Linux 的延迟时间，并且最小延迟时间和平均延迟时间都明显小于 Linux。

3.3.4 多核有压力结果分析

比较结果汇总如下表：

	SpaceChain OS	Linux	比较结果
最大延迟时间（us）	25	121	SpaceChain OS << Linux
最小延迟时间（us）	2	18	SpaceChain OS < Linux
平均延迟时间（us）	5	76	SpaceChain OS < Linux

从统计结果可以看出，在有压力的情况下 SpaceChain OS 的最大延迟时间依然小于 Linux 的延迟时间，并且最小延迟时间和平均延迟时间都明显小于 Linux。

3.3.5 SpaceChain OS 单核与多核结果分析

无压力情况下，SpaceChain OS 在单核和多核下延迟时间的比较结果汇总如下表：

	SpaceChain OS 单核	SpaceChain OS 多核
最大延迟时间（us）	35	13
最小延迟时间（us）	3	2
平均延迟时间（us）	4	4

有压力情况下，SpaceChain OS 在单核和多核下延迟时间的比较结果汇总如下表：

	SpaceChain OS 单核	SpaceChain OS 多核
最大延迟时间（us）	62	25
最小延迟时间（us）	2	2

平均延迟时间（us）	4	5

从结果中可以看出，最大延迟时间并没有因为 CPU 核心数的增加而产生下降，甚至要小于单核的最大延迟时间，平均时间基本一致。这是由于多核开发板的 CPU 的单核性能要高于单核的开发板。

4. 测试结论

在本次的测试过程中主要完成在单核开发板中对 SpaceChain OS、Linux、Linux+RT 进行实时性测试以及多核开发板中对 SpaceChain OS、Linux 进行实时性测试。

在单核无压力情况下，SpaceChain OS 的平均延迟数最低，并且它的最大延迟数与 Linux+RT 相同，都为 35us。说明 Linux+RT 在无压力的情况下实时性与 SpaceChain OS 基本相当，但平均延迟数 SpaceChain OS 要优于 Linux+RT。

在单核有压力情况下，SpaceChain OS 的平均延迟数依然在三个系统中最低，并且它的最大延迟数小于 Linux+RT，说明 SpaceChain OS 的实时性在有压力情况下要优于 Linux+RT。

在多核情况下，SpaceChain OS 在有压力以及无压力情况下实时性都要远优于 Linux 操作系统，并且与单核的 SpaceChain OS 实时性数据进行比较中发现，多核的实时性并没有因为 CPU 核心数的增加而产生下降。