Computer Science and Application 计算机科学与应用, 2022, 12(12), 2937-2949
Published Online December 2022 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2022.1212298

Hans汉斯

一种基于uC/OS-II的任务调度算法及其改进实现

郭艳飞[1]，韩　莹[2]，于　磊[2]

[1]中国化工油气股份有限公司，北京
[2]北方工业大学信息学院，北京

收稿日期：2022年11月22日；录用日期：2022年12月21日；发布日期：2022年12月29日

摘　要

本文在研究不同操作系统任务调度算法的基础上，重点针对嵌入式系统需求，以提高算法的实时性与精简性为目的，提出一种改进方案。改进的调度算法对优先级判定表的冗余部分进行删减，同时引入辅助位图，将原调度算法改进为一种优先级与时间片轮转相结合的调度算法。通过实验分析，该改进算法很好地解决了其空间复杂度较高和最大任务数的瓶颈问题，并具有良好的实时性，不失一般性，对其他系统调度算法设计也具有一定的借鉴意义。

关键词

操作系统，就绪表，优先级，任务调度，位图法

A Task Scheduling Algorithm Based on uC/OS-II and Its Improved Implementation

Yanfei Guo[1], Ying Han[2], Lei Yu[2]

[1]Chemchina Petrochemical Co., Ltd., Beijing
[2]College of Computer Science, North China University of Technology, Beijing

Received: Nov. 22nd, 2022; accepted: Dec. 21st, 2022; published: Dec. 29th, 2022

Abstract

Based on the research of task scheduling algorithms of different operating systems, this paper focuses on the requirements of embedded systems, and proposes an improved scheme to improve the real-time and simplicity of the algorithm. The improved scheduling algorithm deletes the re-

文章引用: 郭艳飞, 韩莹, 于磊. 一种基于 uC/OS-II 的任务调度算法及其改进实现[J]. 计算机科学与应用, 2022, 12(12): 2937-2949. DOI: 10.12677/csa.2022.1212298

dundant part of the priority decision table, and introduces an auxiliary bitmap to improve the original scheduling algorithm, and improves the original scheduling algorithm to combine priority with Round Robin. Through demonstration and analysis, the results show that the improved algorithm solves the bottleneck problem of high space complexity and maximum number of tasks, and has a good reference significance.

Keywords

Operating System, Readiness Table, Priority, Task Scheduling, Bitmap

1. 引言

随着信息技术的发展，实时操作系统凭借其实时性、可靠性等优势被广泛应用于汽车、航空航天、工业自动化和健康医疗设备等高端工业。随着云时代和大数据的到来，工业嵌入式系统功能规模迅速增加，嵌入式系统，如 uC/OS-II，对最大任务数存在限制，有时会在相对复杂和面向对象的系统开发过程中，成为一个瓶颈资源[1] [2]。因此，在保证实时性的基础上提高最大任务数事十分必要的。

uC/OS-II 作为一个嵌入式实时操作系统，包含任务管理与任务调度，具有小巧、实时性强、移植性好、基于优先级的多任务可剥夺等优点[3]。陆伟等人[4]，以 μC/OS-II 操作系统架构为基础，为保证实时性，提出的混合任务调度策。俞佳敏等人[5]基于 uC/OS-II 在原有优先级判断表中，采用分类处理方式，实现最大优先级数扩展。丁宇涛等人[6]选取 μC/OS-II 操作系统为研究对象，分析现有的调度算法，提出两级混合调度策略。

随着嵌入式系统应用的深入，uC/OS-II 对任务数最高 64 个的限制，有时会在相对复杂和面向对象的系统开发过程中，成为一个瓶颈资源。传统 uC/OS-II 在内核调度策略、任务通信机制等方面的不足[7]。划分调度可以比较容易地应用已有的单处理机调度算法，但是系统利用率低。Anderson 等首次提出了半划分调度(Semi-partitioned Scheduling)算法 EDF-fm [8]，提升系统可调度性并相比全局调度降低了上下文切换次数，降低了系统负载。随后，Björn Andersson 等人[9]提出半划分调度算法 EKG，实现了硬实时系统上的可调度性保障。

本文在研究不同操作系统对于任务调度及其优先级算法的基础上，详细分析了嵌入式操作系统中就绪表和优先级表的设计以及获取最高任务优先级的方法，并分析了该方法的优劣。在此基础上，针对其不足之处提出一种改进措施，在保证实时性的基础上减少其空间冗余度，并结合辅助位图扩充了最大任务数，将优先级与时间片相结合，使系统更公平高效。此外，通过比较分析，阐述位图法的优势所在，进一步体会设计者的思想理念。最后对改进方案进行论证，分析其性能和功能上的优越性，为其他研究嵌入式系统的人员有一定的借鉴和指导作用。

2. 三种操作系统的研究

对 Windows 系统、Unix 系统和 Linux 系统进行对比研究，讨论三种不同操作系统下的调度算法及可使用的任务数情况。

对比分析结果如表 1 所示：

Table 1. Comparison table of the three major systems
表 1. 三大系统对比表

	最大任务数	用户可使用的任务数	调度算法	系统类型
Windows 系统	32	31	线程优先级制度 + 时间配额制度	实时 + 分时
Unix 系统	128	77	动态优先级多级反馈循环调度法	分时
Linux 系统	64	56	任务优先级制度	实时

从表格中可以看出，Windows、Unix 和 Linux 系统的采用的调度算法均不相同，但用户可使用的任务数均较少。在 Unix 系统中，采用的动态优先级多级反馈循环调度算法过程复杂，上下文交换等操作繁多，时间开销较大，且确定性难以保证，因而在一定程度上满足不了我们的实时性要求；Linux 系统采用的是基于优先级的任务调度，操作简单，时间确定，满足我们的实时性要求。但若想扩充任务数则显得十分麻烦，系统空间开销也会随之增大，以庞大的空间消耗为代价解决此问题有些得不偿失；对于 Windows 系统，兼顾实时性和分时性，但可以看出，其采用的调度算法最为复杂，用户可用的任务数也极少。

随着大数据和云时代的不断发展，用户对扩充任务数目的需求也随之增长[10]。因此，设计一种可以在保证实时性的基础上扩充任务数目的算法十分重要。通过对这三个系统的分析比较，我们借鉴 Windows 系统和 Unix 系统对进程或任务调度算法思想，将优先级与时间片相结合，在 Linux 系统设计的基础上进行改进，保留该系统原有的实时性，同时达到扩充任务数目的目的。

3. Linux 系统任务调度机制分析

当前，新兴产生的 Linux 嵌入式实时操作系统已经被广泛地应用在各个领域内，对现代化信息技术的发展到至关重要的推动作用[11]。考虑嵌入式系统的实时性，我们主要研究静态调度法。静态调度的目标是把任务分配到各个处理机，并对每一处理机给出所要运行任务的静态运行顺序，即在系统调度前就确定各个任务的优先级。静态调度算法实现简单，调度的额外开销小，在系统超载时可预测性好。但也具有很大的局限性，例如资源利用率低、受系统支持的优先级个数限制以及灵活性和自适应性差等。

下面分析其相关方面的具体设计。

3.1. 就绪表的设计

以 uC/OS-II 为例，μC/OS-II 任务优先级调度算法通过 OSUnMapTbl 逆映射表直接查表获得当前就绪任务的最高优先级[12]，在建立任务时，系统就会给任务分配一个优先级号，优先级号越低，任务的优先级越高。就绪表中存放着任务的就绪标志，该表有两个变量就绪组 OSRdyGrp 和就绪表 OSRdyTbl[]。就绪组 OSRdyGrp 是个 8 bit 的数值，就绪表 OSRdyTbl[]是一维数组。

就绪表和就绪组的关系如图 1 所示。

从图 1 中可以看出，在 OSRdyGrp 中，任务按照优先级分组，8 个任务为一组，如优先级为 8~15 的任务就处于第 1 号就绪组，即表示为 00000010。反之亦然：若 OSRdyGrp 是 00010010，则说明 OSRdyTbl[]中第 1 组和第 4 组中有任务处于就绪态，即优先级为 8~15 和 32~39 中有任务处于就绪态，具体是哪个任务处于就绪状态，还需要查看 OSRdyTbl[]中对应组中哪一位置为 1。

使任务进入就绪态的代码如下：

```
OSRdyGrp    |=    OSMapTbl[prio>>3];
OSRdyTbl[prio>>3]    |=    OSMapTbl[prio& 0x07];
```

从上面两个式子可以看出，任务优先级的低三位用于确定任务在就绪组 OSRdyGrp 中的所在位。接下去的三位用于确定是在 OSRdyTbl[]数组的第几个元素。OSMapTbl[]是在 ROM 中的屏蔽字，用于限制 OSRdyTbl[]数组的元素下标在 0 到 7 之间。

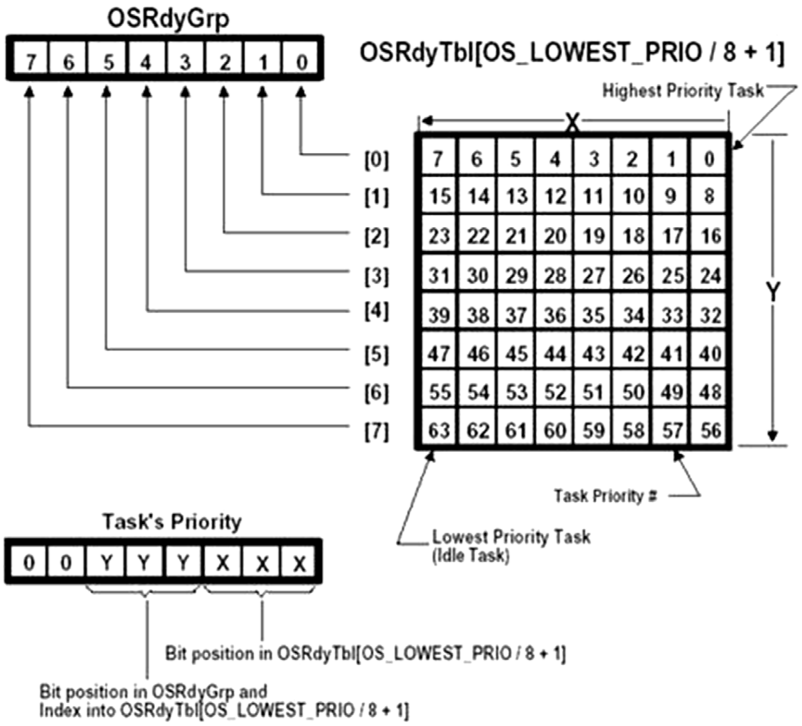

Figure 1. Diagram of readiness groups and readiness tables
图 1. 就绪组和就绪表的关系图

OSMapTbl[Index]如表 2 所示：

Table 2. OSMapTbl[Index] table
表 2. OSMapTbl[Index]表

Index	BitMask(Binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

此外，还要再次提醒读者，就绪表 OSRdyTbl[]并不是个二维数组，图 1 中画成二维表只是为了更好地描述就绪表和就绪组之间的关系，同时更直白地表示出就绪表中每位和优先级的映射关系。

3.2. 最高优先级算法

上面介绍了任务调度过程和就绪表相关内容，可以发现，这两部分都和优先级密不可分，因而，如何在较短时间内获得最高优先级号变得尤为重要。

3.2.1. 几种获取最高优先级方法的比较与分析

任务的优先级是在创建任务时就确定了的。获取最高优先级的方法很多[13] [14] [15]，比如：

1) 通过将就绪任务的优先级两两进行比较，从而得到最高优先级；

2) 按照优先级对任务进行降序排列，第一个任务的优先级即所求；

3) 从就绪表第 0 位开始进行顺序查找，找到的第一个处于就绪态的任务的优先级即为最高优先级。

比较与分析：

如果任务数目极少，以上提到的三种方法均可行。但是任务数目较多时，上述方法的时间复杂度不确定或时间开销太大，难以保证实时性，与嵌入式系统设计初衷相悖。因而提出以空间换时间的方法，引入优先级判定表 OSUnMapTbl[]，利用位图法，通过查表获得最高优先级任务。

3.2.2. 位图法

位图(Bitmap)，即用每一位来存放某种状态。通过将数组下标与应用中的一些值关联映射，数组中该下标所指定的位置上的元素可以用来标识应用中值的情况，简言之，下标即索引。位图索引可以利用多核和多处理器系统，处理随时间增加但不经常更改的数据，在对低基数数据的查询中优势明显[16]。

例如：用 1 bit 便可标识性别，如令 1 表男性，0 代表女性。

基于位图表示的索引技术对于有效地解决复杂和特殊查询非常有用，而且不需要添加额外的硬件。它们通过利用低成本的布尔运算和多个索引扫描来显著地改善查询处理时间[17]。此外，位图数组中每个元素在内存中占用 1 位，比原始数据需要更少的空间[18]，减少系统空间开销。以上两点均体现出位图法的优越性。

实时性的核心含义在于确定性，而不仅仅是速度快。将位图法应用到嵌入式系统中的进程调度及事件管理，可使得系统能够快速找到处于就绪态的优先级最高的任务，速度极快而且运行时间是确定的，从而提高了系统的响应速度和实时性[19]。

3.2.3. 计算优先级的具体方法

优先级判定表如下图 2 所示：

```
INT8U  const  OSUnMapTbl[256] = {
    0u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x00 to 0x0F
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x10 to 0x1F
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x20 to 0x2F
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x30 to 0x3F
    6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x40 to 0x4F
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x50 to 0x5F
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x60 to 0x6F
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x70 to 0x7F
    7u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x80 to 0x8F
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0x90 to 0x9F
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xA0 to 0xAF
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xB0 to 0xBF
    6u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xC0 to 0xCF
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xD0 to 0xDF
    5u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, /* 0xE0 to 0xEF
    4u, 0u, 1u, 0u, 2u, 0u, 1u, 0u, 3u, 0u, 1u, 0u, 2u, 0u, 1u, 0u  /* 0xF0 to 0xFF
};
```

Figure 2. Prioritization table
图 2. 优先级判定表

事实上，优先级判定表也是一个一维数组，我们将其写成二维表的形式，只是为了方便说明问题和查找计算，这一点要注意。可以看出，映射表在系统中为常量，通过该表，查找最高就绪优先级的算法就可以巧妙地回避遍历操作，从而降低时间复杂度(降为 O(1))。

计算优先级的相关代码为：

1) i = OSUnMapTbl[OSRdyGrp];//找到行号

2) j = OSUnMapTbl[OSRdyTbl[i]];//找到列号

3) TaskPrior = 8*i+j =(i<<3)+ j;//找到最高优先级号

代码(1)是指将就绪组的值作为索引，在优先级判定表中寻找到最高优先级在就绪表 OSRdyTbl[]中对应的行号；

代码(2)是指将就绪表中第 i 行数据作为索引，在优先级判定表中寻找到最高优先级在就绪表 OSRdyTbl[]中对应的列号；

代码(3)提供了两种计算最高优先级号的方法：

1) TaskPrior = 8*i+j；

由步骤(1)和步骤(2)可得最高优先级号在就绪表中的行号和列号，对照就绪表图所示的排列(8 位为一组)，易得出 8*i+j 即为所求的最高优先级号。

2) TaskPrior = (i<<3)+ j；

该方法实际上与方法 1)相同，只是用移位代替了乘除(在二进制中，i<<3 表示将 i 值左移三位，即乘 8)。

下面我们举例说明上述计算方法和相关代码：

例子：假设 OSRdyGrp=98(d)=01100010(b)。

步骤 1：以 OSRdyGrp 的值为索引，在优先级判定表(OSUnMapTbl[])中查找对应数字作为行号。

这里需要注意：98 是十进制的，要将其先化为十六进制 0x62。然后锁定注释为：/*0x60 to 0x6F*/行，该行表示十六进制下从 0x60 到 0x6F。接着，按从左往右的方向，将第一个数字作为 0x60 对应的号，那么 0x62 在表中对应的数字就是 1u。所以 i=1，即行号是 1。

步骤 2：得到 OSRdyTbl[i]对应的值并计算列号。

i 值第一步已得出，所以 OSRdyTbl[i]=OSRdyTbl[1]。假设此时 OSRdyTbl[1]=01100000(b)=96(d)。那么就在 OSUnMapTbl[]表中查找第 96 位的数字，查找方式和第一步相同。可以得到第 96 位对应的数值为 5u。所以 j=5，即列号是 5。

步骤 3：计算最高优先级任务对应的优先级。

i<<3+j = 1*8+5 = 13。

即在就绪表中最高优先级任务对应的优先级为 13。

验证：

OSRdyGrp=98(d)=01100010(b)，则在第一组、第五组和第六组中均有任务处于就绪状态。

OSRdyTbl[1]=01100000(b)，即就绪表中第一行第五列和第六列所代表的任务处于就绪态。如图 3 表格中[1]所示橘色部分(0 表示未就绪，1 表示就绪)。

这里我们假定 OSRdyTbl[5]=OSRdyTbl[6]=11111111；即第五组和第六组所有任务都处于就绪态。如图 3 格中[5] [6]所示绿色部分。

从图 3 我们可以很容易看出来最高优先级就是[1]行 5 列所代表的优先级，即 13 (优先级号越小，优先级越高)。得证。

	7	6	5	4	3	2	1	0
[0]	0	0	0	0	0	0	0	0
[1]	0	1	1	0	0	0	0	0
[2]	0	0	0	0	0	0	0	0
[3]	0	0	0	0	0	0	0	0
[4]	0	0	0	0	0	0	0	0
[5]	1	1	1	1	1	1	1	1
[6]	1	1	1	1	1	1	1	1
[7]	0	0	0	0	0	0	0	0

Figure 3. Readiness table example diagram
图 3. 就绪表实例图

注意

读者要明确一点：查找最高优先级号，我们最终的落脚点是就绪表。即是在就绪表中找到最高优先级任务，优先级判定表只是方便我们锁定最高优先级所在位置的一个手段。

优先级、就绪表、就绪组和优先级判定表之间的关系图如图 4。

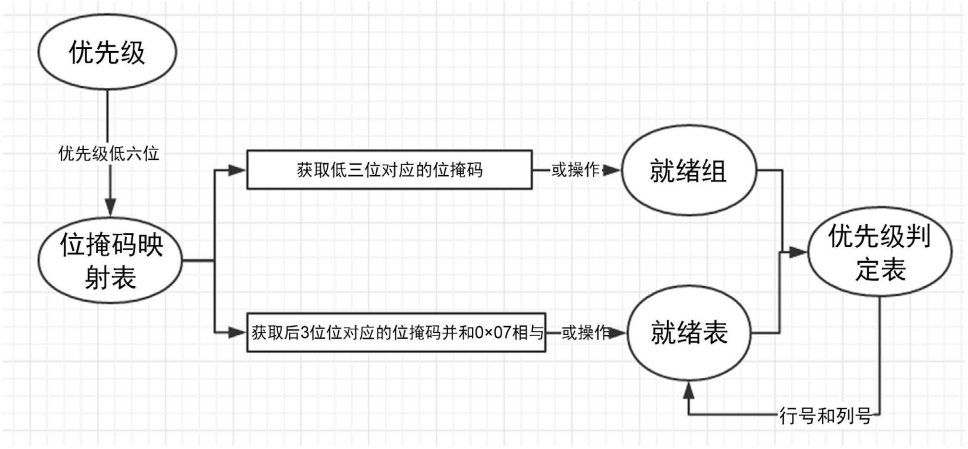

Figure 4. Relationship diagram
图 4. 关系图

4. 改进方案的设计与实现

在嵌入式系统中，每个任务的优先级是唯一的，用户能够使用的任务数量也是有限的。如果想扩充任务数目，同时又希望保证实时性，只能再扩大优先级判定表的空间，导致系统空间消耗逐渐增大。因而需要改善原优先级判定表，同时保证不破坏其实时性。

4.1. 对优先级判定表的分析及优化

1) 优点：

引入优先级判定表，以空间换时间，借助位图法的优势使时间复杂度变小，从而可以更快速地获取最高优先级任务，最大程度满足操作系统的实时性要求。

2) 缺点：

观察优先级判定表发现，该表存在很大的冗余：除了第一列不同外，其他列都是相同的，在一定程度上也是在浪费空间资源。因此，对优先级判定表进行改进十分必要。

3) 改进方案：

针对上文分析的优先级判别表的不足之处，在保证实时性的条件下考虑对该表进行精简，舍去部分冗余，同时引入辅助位图，将优先级调度法和时间片轮转法相结合，解决任务数目扩充问题。

对优先级判定表处理方法如下：

a) 将原优先级判别表按图 3 所示方式看成 16*16 的二维表，横纵下标均从 0 开始，然后提取出原优先级判定表的第 0 行和第 0 列；

b) 分析第 0 行和第 0 列的下标值发现，第 i 行奇数下标所对应的数值恒为 4，第 j 列奇数下标所对应的数值恒为 0，冗余度依然很大，因而再次提取相同部分；

c) 将 8 bit 的优先级数值拆分乘高四位和低四位分别检查，在检查时，首先将这四位与 0001(b)进行“与”操作，检验其奇偶性。若为奇数，则行或列的值取 0 (低四位)或 4 (高四位)；若为偶数，则将该四位右移一位，然后查表 1 得到其相应的数值作为行号或列号。

改进后的优先级判别表如表 3 所示：

Table 3. Improved priority discrimination table
表 3. 改进后的优先级判别表

下标	0	1	2	3	4	5	6	7
低 4 位=0	0	5	6	5	7	5	6	5
低 4 位！=0	0	1	2	1	3	1	2	1

辅助位图表如表 4 所示：

Table 4. Auxiliary bit charts
表 4. 辅助位图表

0	0

优先级计算对应的代码段修改为：

```
Char    OSUnMapTblChanged_1[] = {0,5,6,5,7,5,6,5};/*低 4 位=0 时对应的表；*/
Char    OSUnMapTblChanged_2[] = {0,1,2,1,3,1,2,1};/*低 4 位!=0 时对应的表；*/
/*行号的确定*/
judge_odd = OSRdyGrp& 0x01; /*判断低四位的奇偶性*/
if(0 == judge_odd) /*若低 4 位为偶数*/
judge_zero = (OSRdyGrp>>1) & 0x07;/*判断低 4 位是否为 0*/
if(0 == judge_zero) /*低 4 位为 0*/
/*继续查看高四位，方法同上*/
else /*低 4 位不为 0*/
row = OSUnMapTblChanged_2[judge_zero]; /*查低 4 位不为 0 对应的表*/
else/*低四位是奇数*/
row = 0;
```

列号的确定方式同行号，此处不再赘述。

检验：(仍采用上文的例子)

OSRdyGrp=98(d)=01100010(b)，低 4 位为 0010(b)，与 0001(b)进行与操作，如图 5 所示：

```
    0010
&   0001
-----------
    0000   ← 等于 0，说明原数为偶
```

Figure 5. Low 4-bit parity judgmen
图 5. 低 4 位奇偶性判断

原数为偶数，则将其右移一位，得：0001(b) = 1(d)，查表 2，下标为 1 且低 4 位!=0 的数值为 1，即在就绪表中对应的行号为 1。

同理可以求得 OSRdyTbl[1]= 01100000(b)在就绪表中对应的列号为 5，检验正确。

4.2. 扩充任务数目方案设计

若是简单地通过扩充就绪组和就绪表的长度，如：将 OSRdyGrp 由原来的 INT 8U 扩展成 INT 16U，则任务数从之前的 64 个增到 128 个，但优先级判定表也随之翻倍增长。因而引入辅助位图表来达到同样的效果，如表 4 所示。

每一个优先级都有其对应的辅助位图。引入辅助位图之后，优先级号不再是任务的唯一标识，即系统允许不同的任务有相同的优先级号。这种情况下，就绪表的每一位是否置为“1”取决于该优先级下的辅助位图表是否全为“1”(“1”表示空间已被分配，“0”表示空间未被分配)。

我们设计的辅助位图只有 2 bit，一方面是考虑 128 个任务数目足以满足用户需求；另一方面是考虑尽可能地减少空间上的浪费。

系统在给任务分配优先级号的时候，若就绪表中对应位置为 1，则考虑下一位；若为 0，则依次将辅助位图中的空闲空间分配出去，并将其下标作为辅助位图标识，结合优先级号，作为该任务的唯一标识。

系统进行任务调度时，对同优先级任务下的不同任务采用时间片轮转法。设置两个标志，分别标志两个任务的运行状态，若运行时间到达时间片所规定的时间额度，就将任务设为挂起状态，同时，将另一个同优先级任务设为运行状态。如此便将优先级调度与时间片轮转调度法结合起来，高效且不失公平。

具体任务调度示意图如图 6：

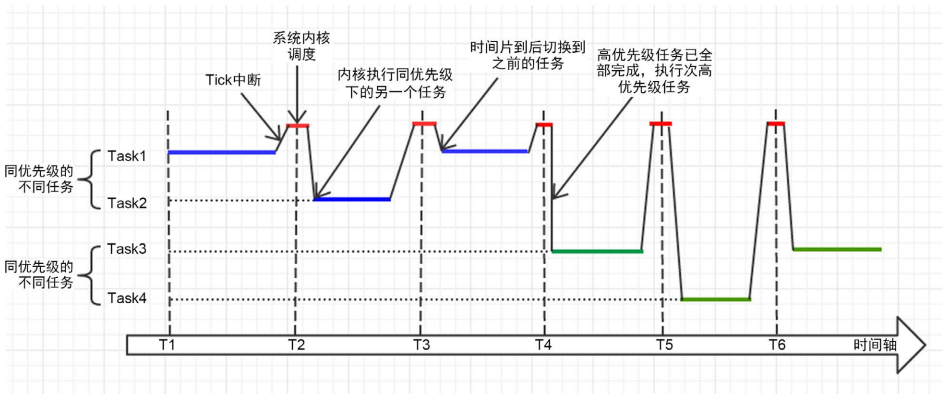

Figure 6. Task scheduling schematic
图 6. 任务调度示意图

在代码方面，我们只需要设置一个指针，标志辅助位图的状态；然后再在代码中加入 if 判断即可实现。

5. 性能分析

空间资源方面

改进后空间节约率算法为：

$$P = \frac{M_0 - M_1 - C*n}{M_0} * 100\%$$

其中，P 为空间节约率，M_0 为原空间占用量，这里为常数 256，M_1 为改进后占用的空间量，值为常数 16，C 表示 INT 8u，即 8 bit，n 指辅助位图占用的比特数。

从公式中可以看出，P 随 n 的增大而减小，二者成反比例。因而取 $n=2$ 时，P 可以得到最大值，即最为节约空间($n \neq 1$，否则与原设计无差别)。这就是我们改进算法中将辅助位图表取 2 位的原因之一。

原优先级判定表为 16*16 = 256 Byte，改进后为 2*8 = 16 Byte，空间使用减少的百分比为：

$$P_1 = \frac{256-16}{256} * 100\% = 93.75\%$$

此外，辅助位图所耗费的空间开销为：64*2 = 128 Byte；

综合以上两个方面，求得的空间节约率为：

$$P_2 = \frac{256-16-128}{256} * 100\% = 56.25\%$$

可以看出，改进后的算法很大程度减少了空间资源的浪费，同时也将支持的最大任务数从之前的 64 个扩充到了 128 个，基本满足用户需求。

5.2. 实时性方面

查找最高优先级任务采用的方法仍是位图法，且代码中只使用到了条件判断、少量的逻辑运算和移位操作，与原代码相比，运行时间几乎没有差别，实时性得到了保证。

1) 平均周转时间

平均周转时间算法为：

$$T = \frac{\sum_i^n t_i}{n}$$

其中，T 为平均周转时间，i 为第几个任务，n 为任务总数，t_i 为每个任务的周转时间。

假设有 5 个批处理作业(A、B、C、D 和 E)几乎同时到达一个计算中心，估计运行时间分别为 3、6、6、9、3 分钟，它们的优先级分别为 5、4、3、2、1 (1 为最高优先级)。

i) 最高优先级算法，如表 5 所示：

Table 5. Turnaround time for each task of the highest priority algorithm
表 5. 最高优先级算法各任务周转时间

开始时间	结束时间	所需时长	任务号
0 min	3 min	3 min	E
3 min	12 min	9 min	D

Continued

12 min	18 min	6 min	C
18 min	24 min	6 min	B
24 min	27 min	3 min	A

由表知：各任务周转时间分别为：27、24、18、12、3；

平均周转时间为：T_1 = 16.8 min。

ii) 时间片轮转算法(假设时间片为 3 min)，如表 6 所示：

Table 6. Time slice rotation method turnaround time for each task
表 6. 时间片轮转法各任务周转时间

开始时间	结束时间	所需时长	任务号
0 min	3 min	3 min	E
3 min	6 min	3 min	A
6 min	9 min	6 min	C
9 min	12 min	6 min	B
12 min	15 min	9 min	D
15 min	18 min	6 min	C
18 min	21 min	6 min	B
21 min	27 min	9 min	D

由表知，各任务周转时间分别为：6、21、18、27、3；

平均周转时间为：T_2 = 15 min。

iii) 最高优先级算法与时间片轮转算法相结合，如表 7 所示。

不妨将原优先级调为：2、4、2、3、1。

Table 7. Priority combined with time slice for each task turnaround time
表 7. 优先级与时间片相结合各任务周转时间

开始时间	结束时间	所需时长	任务号
0 min	3 min	3 min	E
3 min	6 min	3 min	A
6 min	12 min	6 min	C
12 min	21 min	9 min	D
21 min	27 min	6 min	B

由表知，各任务周转时间为：6、27、12、21、3；

平均周转时间为：T_2 = 13.8 min。

在三种调度方式下计算任务平均周转时间，对比发现，改进后的算法平均周转时间比单独使用优先级算法或者时间片轮转算法用时更短。当然，在某些情况下，改进算法用时居于另外两种算法之间，但

综合来看，改进算法同时考虑了任务优先级和公平性，并且平均周转时间并不长，执行效率更高。

2) 系统吞吐率

以上面例子为准，我们研究在 20 min 内，系统的吞吐率。每个任务完成度以其已完成时间份额为准。计算公式为：

$$R=\frac{\sum_i^n S_i*\frac{1}{N_i}}{T'}$$

其中，i 表示第几个任务，n 为任务总数，N_i 为第 i 个任务完成所需时间，S_i 为第 i 个任务当前已完成部分所用的时间，T' 为研究的时间范围，这里为定值 20。

则在最高优先级调度法下：$R_1=\frac{3*\frac{1}{3}+6*\frac{1}{6}+9*\frac{1}{9}+3*\frac{1}{6}}{21}=16.7\%$；

时间片调度算法下：$R_2=\frac{3*\frac{1}{3}+3*\frac{1}{3}+6*\frac{1}{6}+6*\frac{1}{6}+3*\frac{1}{9}}{21}=20.6\%$；

改进算法下：$R_3=\frac{3*\frac{1}{3}+3*\frac{1}{3}+6*\frac{1}{6}+9*\frac{1}{9}}{21}=19.0\%$。

从以上计算中可以看出，改进算法相较单纯的最高优先级调度法有更高的系统吞吐率，且与时间片调度算法相差不大。

综上可以看出，无论是实时性还是系统的空间利用率和吞吐率，改进算法都有着很大的优势。

6. 结束语

本文系统地阐述了 Windows 系统、Unix 系统和 Linux 系统任务调度相关内容，基于嵌入式实时操作系统 uC/OS-II，结合优先级与时间片轮转相结合改进调度算法，解决了其空间复杂度较高和最大任务数的瓶颈问题。其中，重点介绍了嵌入式系统中就绪表相关知识和优先级算法及其代码表示。在此基础上，通过举例，逐步进行最高优先级的求解，更深层地理解最高优先级的算法。求解后对该算法进行了验证，并给出了几个表之间的关系图，方便读者更好地理解其内在关系。此外，还对几种求解优先级的方法进行了对比分析，对位图法进行了相关描述，体现出引入优先级判定表的必要性和优越性。然后简要分析了嵌入式系统优先级算法的优缺点，并针对优先级判定表的冗余问题和任务数目扩充问题提出解决方案，最后通过性能分析得出，该算法很大程度上减少了空间复杂度，同时保证了实时性。本文对研究嵌入式操作系统，尤其是想要深入理解其优先级机制的人员有借鉴和指导意义。

参考文献

[1] 邢红星, 魏叶华, 乐懿. 硬件成本缩减的异构分布式嵌入式系统调度算法[J]. 计算机工程与科学, 2021, 43(2): 258-265.

[2] 阚宏伟, 马小平, 杜林. uC/OS-II 任务数扩充的理论与实现[J]. 计算机工程, 2007, 33(13): 99-100+103.

[3] 潘佳腾. VC6.0 的 μC/OS-II 移植可行性分析研究[J]. 单片机与嵌入式系统应用, 2018, 18(3): 19-22+28.

[4] 陆伟, 张龙妹. 嵌入式操作系统混合任务调度技术与策略研究[J]. 计算机工程与应用, 2015, 51(15): 6-11.

[5] 俞佳敏, 王成群, 徐伟强. 实时操作系统 μC/OS-II 最大优先级数扩展实现[J]. 无线电通信技术, 2018, 44(6): 550-553.

[6] 丁宇涛. 基于 FPGA 的 μC/OS-II 操作系统任务调度算法的研究与实现[D]: [硕士学位论文]. 哈尔滨: 哈尔滨理工

大学, 2022. https://doi.org/10.27063/d.cnki.ghlgu.2022.000802

[7] 陈刚, 关楠, 吕鸣松, 王义. 实时多核嵌入式系统研究综述[J]. 软件学报, 2018, 29(7): 2152-2176. https://doi.org/10.13328/j.cnki.jos.005580

[8] Anderson, J.H., Bud, V. and Devi, U.C. (2005) An EDF-Based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. 17*th Euromicro Conference on Real-Time Systems* (*ECRTS*'05), Balearic Islands, 6-8 July 2005, 199-208. https://doi.org/10.1109/ECRTS.2005.6

[9] Andersson, B. and Tovar, E. (2006) Multiprocessor Scheduling with Few Preemptions. 12*th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (*RTCSA*'06), Sydney, 16-18 August 2006, 322-334. https://doi.org/10.1109/RTCSA.2006.45

[10] Panda, S.K. and Jana, P.K. (2019) An Energy-Efficient Task Scheduling Algorithm for Heterogeneous Cloud Computing Systems. *Cluster Computing*, **22**, 509-527. https://doi.org/10.1007/s10586-018-2858-8

[11] 李欣, 白兴武. 基于 Linux 的嵌入式实时操作系统任务调度算法优化[J]. 自动化与仪器仪表, 2020(9): 48-51. https://doi.org/10.14016/j.cnki.1001-9227.2020.09.048

[12] 黄国兵, 李瑞玲, 李华丽, 王琼. μC/OS-II 任务优先级调度算法分析与改进[J]. 计算机工程, 2015, 41(8): 52-54+60.

[13] Pathan, R., Voudouris, P. and Stenstrom, P. (2018) Scheduling Parallel Real-Time Recurrent Tasks on Multicore Platforms. *IEEE Transactions on Parallel and Distributed Systems*, **29**, 915-928. https://doi.org/10.1109/TPDS.2017.2777449

[14] Lee, H., Roh, J. and Seo, E. (2018) A GPU Kernel Transactionization Scheme for Preemptive Priority Scheduling. 2018 *IEEE Real-Time and Embedded Technology and Applications Symposium*, Porto, 11-13 April 2018, 202-213. https://doi.org/10.1109/RTAS.2018.00029

[15] Lin, J.S., Huang, C.Y. and Fang, C.C. (2022) Analysis and Assessment of Software Reliability Modeling with Preemptive Priority Queueing Policy. *Journal of Systems and Software*, **187**, Article ID: 111249. https://doi.org/10.1016/j.jss.2022.111249

[16] Yildiz, B. (2021) Optimizing Bitmap Index Encoding for High Performance Queries. *Concurrency and Computation*: *Practice & Experience*, **33**, e5943. https://doi.org/10.1002/cpe.5943

[17] Vanichayobon, S., Manfuekphan, J. andGruenwald, L. (2006) Scatter Bitmap: Space-Time Efficient Bitmap Indexing for Equality and Membership Queries. 2006 *IEEE Conference on Cybernetics and Intelligent Systems*, Bangkok, 7-9 June 2006, 1-6. https://doi.org/10.1109/ICCIS.2006.252354

[18] Rajurkar, J. and Khan, T.K. (2015) Efficient Query Processing and Optimization in SQL Using Compressed Bitmap Indexing for Set Predicates. 2015 *IEEE* 9*th International Conference on Intelligent Systems and Control*, Coimbatore, 9-10 January 2015, 1-5. https://doi.org/10.1109/ISCO.2015.7282354

[19] Yildiz, B. (2019) High Performance Queries Using Compressed Bitmap Indexes. In: Schwardmann, U., Boehme, C., Heras, D.B., *et al*., Eds., *Euro-Par* 2019: *Parallel Processing Workshops*. *Euro-Par* 2019. *Lecture Notes in Computer Science*, Springer, Cham, 493-505. https://doi.org/10.1007/978-3-030-48340-1_38