

μC/OS-II 优先级反转与死锁问题的解决

彭磊 韩忠东 马华 马晓艳

(泰山医学院信息工程学院 山东 泰安 271016)

摘要 μC/OS-II 没有真正实现优先级继承协议解决优先级反转,也没有提供有效的死锁解决方法。对任务管理机制改进后,扩展了同优先级任务的时间片轮转调度算法,实现了真正的优先级继承协议;并且使用资源请求、分配矩阵来表示资源分配情况,在任务申请资源阻塞时进行死锁的检测与解除。通过性能分析与测试验证证明了改进算法的有效性和实时性。

关键词 μC/OS-II 优先级反转 优先级继承协议 死锁

中图分类号 TP316 文献标识码 A

SOLVING PRIORITY INVERSION AND DEADLOCK PROBLEMS IN μC/OS-II

Peng Lei Han Zhongdong Ma Hua Ma Xiaoyan

(College of Information Engineering, Taishan Medical University, Taian 271016, Shandong, China)

Abstract μC/OS-II does not realise the true priority inheritance protocol to resolve priority inversion. It does not provide an effective solution to solving deadlock either. After improving the task management mechanism, the same-priority-task time slice circular scheduling algorithm is expanded to realise the true priority inheritance protocol. Resource allocation is represented by used resource request and allocation matrix. When a task requesting a resource is blocked, its deadlock is automatically detected and resolved. Through performance analysis and test validation, the effectiveness and timeliness of the improved algorithm is proven.

Keywords μC/OS-II Priority inversion Priority inheritance protocol Deadlock

0 引言

μC/OS-II 是 Jean J. Labrosse 编写的基于优先级抢占式的实时多任务操作系统,包含了实时内核、任务管理、时间管理、任务间通信与同步和内存管理等功能^[1]。在嵌入式领域已得到广泛应用,但在具体应用中存在以下不足:

(1) 无法采用真正的优先级继承协议解决优先级反转 优先级继承协议的实现需要操作系统本身支持同优先级任务,而 μC/OS-II 不支持同优先级任务,因此需要程序员给互斥信号量设定一个较高的固定不变的优先级作为继承优先级 PIP,占用资源的任务被阻塞时将自身优先级提升到 PIP,释放资源时再回到自身优先级。这就要求程序员要了解 μC/OS-II 的内核机制并参与优先级反转问题的解决。

(2) 没有提供有效的死锁解决方法 在多任务系统中,由于竞争资源或任务间推进顺序不当会引起死锁。在较简单的嵌入式应用系统中一般不易出现,但随着应用越来越复杂,死锁已成为一个无法避免的问题。μC/OS-II 内核没有提供有效的死锁解决方法,而是要求程序员通过定义信号量超时或其他方式来解决。

通过以上分析,说明了只有对 μC/OS-II 的内核任务管理机制做出改进,使其支持同优先级任务,从而实现优先级继承协议,同时增加死锁的检测与解除机制才能更好地满足实际应用的需要。

1 优先级反转问题的解决

1.1 同优先级时间片轮转算法

μC/OS-II 的每一个任务都有自己的优先级,不能和其他任务相同,即 μC/OS-II 将任务的优先级当作任务的标识来使用^[2]。为了支持同优先级任务,不能再用优先级唯一标识任务,因此使用任务编号 OSTCBId 来标识任务。OSTCBId 为 16 位无符号整型,其中 0~3 位标识同优先级内任务编号,4~11 位标识任务优先级,即任务编号等于任务优先级左移 4 位加优先级内编号 OSTCBId = OSTCBPrio << 4 + OSTCBPrioId。

任务就绪表中的 OSRdyGrp、OSRdyTbl 为 16 位无符号整型,另外增加了同级任务就绪表 OSRdySlcTbl[][16]和同级任务就绪备份表 OSRdySlcBckTbl[][16]。如图 1 所示。

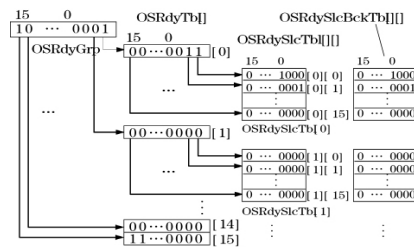


图 1 任务就绪表

收稿日期:2010-05-09。彭磊,讲师,主研领域:人工智能,操作系统。

为了实现同优先级任务时间片轮转算法,需要定义一个时间片大小的宏,即一个时间片包含的节拍数,这个数是大小可调的: #define OS_TICKS_PER_SLICE 10。

任务控制块 TCB 中还需要增加以下几项:

```
INT8U   OSTCBBitZ;
INT8U   OSTCBSlice;
INT8U   OSTCBSliceCnt;
```

OSTCBBitZ 表示任务在同级就绪任务表 OSRdySlcTbl 与备份表 OSRdySlcBckTbl 中位的位置。OSTCBSlice 表示任务一个时间片所占节拍数,即等于 OS_TICKS_PER_SLICE。OSTCBSliceCnt 表示任务当前时间片剩余节拍数。

时间片轮转算法中时间片的计数与轮转调度是通过时钟节拍服务函数 OSTimeTick() 来实现的。在函数中增加对当前执行态的任务 TCB 中的时间片剩余值 OSTCBSliceCnt 进行减一的操作。当 OSTCBSliceCnt 减到 0 时,说明该任务的时间片已用完,把当前任务在就绪表 OSRdySlcTbl 中对应的标志位置 0,调度同优先级任务中的下一个任务执行,以此类推。如果 OSRdySlcTbl 表中本优先级的任务标志都已置 0,说明本级任务都已完成了一个时间片,即本级任务时间片轮转一次。于是将 OSRdySlcBckTbl 表中本级任务的备份标志复制回 OSRdySlcTbl 表中,同时将 TCB 中的 OSTCBSlice 赋值给 OSTCBSliceCnt,再进行任务调度,开始新一轮的时间片轮转。

1.2 优先级继承协议的改进

解决优先级反转有许多种方法。其中普遍使用的有优先级继承协议和优先级天花板协议。

优先级继承协议是指当高优先级任务 Task1 申请低优先级任务 Task2 所占用的资源 R 时,把 Task2 的优先级提高到与 Task1 等同的优先级 PIP。当 Task2 释放资源 R 时,立即将其优先级降回到原来的优先级。

优先级天花板协议是指每个资源 R 与一个天花板优先级 PCP 绑定,PCP 是所有将要使用 R 的任务中最高优先级。当任务 Task1 申请并得到资源 R 时,立即把 Task1 的优先级提升到 PCP。释放资源 R 时,将其优先级降回到原来的优先级。

二者的本质区别是:优先级继承协议只有当占用资源的任务被阻塞时,才提高任务的优先级。而天花板协议是任务申请到资源立即提升任务的优先级。

$\mu\text{C}/\text{OS-II}$ 采用的是准优先级继承协议,因为不支持同优先级任务,所以创建信号量时需人工指定一个较高的优先级作为继承优先级 PIP^[4],即 OSMutexCreate(INT8U pip, INT8U * err)。改进的算法已经支持同优先级任务,因此可以真正地实现优先级继承协议,无需人工指定 PIP,简化了编程接口。修改如下:

(1) 创建信号量时无需给资源赋继承优先级 PIP,函数接口改为: OS_EVENT * OSMutexCreate(INT8U * err)。

(2) 低优先级任务 Task2 得到资源 R 后不提高优先级,当有高优先级任务 Task1 申请 R 时才将 Task2 的优先级提升到 PIP 优先级。PIP 是系统自动获取的,首先检查与 Task1 同级的任务 ID 是否有空闲,如果有则将其优先级作为 PIP,提升 Task2 到 PIP,同时获得空闲任务 ID,否则找高一级的空闲任务 ID。查找空闲任务 ID 是通过搜索 OSTCBPrioTbl 表获得的。算法如下:

```
for( i = highId | 0x0F; i >= 0; i-- ) {
    if( OSTCBPrioTbl[i] == ( OS_TCB* ) 0 ) {
        pipId = i; break;
    }
```

```
}
}
```

(3) 任务释放资源时如果优先级已经提高到了 PIP,则返回到原来的优先级,再进行任务调度。

2 死锁问题的解决

2.1 死锁检测与解除算法

引入死锁检测机制要注意检测时机的选择。死锁检测过于频繁会增加系统的开销,影响系统效率;检测不及时,可能会使多个任务卷入死锁,降低处理机等资源的利用率。因此算法选择在任务申请资源 R 无法满足,同时占用资源 R 的任务也处于等待状态时进行死锁的检测,这时很有可能处于死锁状态。即在 OSMutexPend() 函数中实现死锁检测。

死锁的检测算法需要使用资源分配图 RAG(Resource Allocation Graph) 来记录系统中资源的请求、分配状况,用矩阵 M 来表示 RAG。

$$M = \begin{bmatrix} e_{11} & e_{12} & \cdots & e_{1n} \\ e_{21} & e_{22} & \cdots & e_{2n} \\ \cdots & \cdots & e_{ij} & \cdots \\ e_{m1} & e_{m2} & \cdots & e_{mn} \end{bmatrix}$$

矩阵 M 的行表示任务节点,列表示资源节点, e_{ij} 表示第 i 个任务对第 j 种资源的请求、分配状况,取值范围是 { request, allocation }。取 request 表示 RAG 中的请求边,取 allocation 表示 RAG 中的分配边。取 0 表示任务和资源之间没有请求、分配关系。为了节省存储空间,可以使用 2 个二进制位来表示 e_{ij} ,因此将 M 分解成两个矩阵:资源请求矩阵 MatrixR 和资源分配矩阵 MatrixA。

$$\text{MatrixR} = \begin{bmatrix} 0 & \cdots & 0 & r_{1n} & \cdots & r_{12} & r_{11} \\ 0 & \cdots & 0 & r_{2n} & \cdots & r_{22} & r_{21} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & r_{mn} & \cdots & r_{m2} & r_{m1} \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ \cdots \\ r_m \end{bmatrix}$$

MatrixR 是 $m \times 8$ 矩阵, m 表示总任务数量, n 表示资源数量,且 $n \leq 8$ 。如果 $n > 8$, MatrixR 可定义为 $m \times 16$ 或 $m \times 32$ 矩阵。其中 r_{ij} 是一个二进制位,取值范围是 {0, 1}, 取 1 表示 RAG 中存在请求边,取 0 表示不存在请求边,因此可以用一个整数 r_i 来代替 MatrixR 的一行。即定义一维数组 INT8U MatrixR[m] 来表示矩阵 MatrixR。MatrixR[i] ($0 \leq i < m$) 表示第 i 个任务,其中的每个二进制位表示一种资源的请求状况,只使用右 n 位,左 8-n 位不使用,补 0。

$$\text{MatrixA}' = [a_1 \ a_2 \ \cdots \ a_m]$$

MatrixA 也是 $m \times 8$ 矩阵,结构与 MatrixR 类似,可定义数组 INT8U MatrixA[m] 来表示。且 MatrixR[i] 和 MatrixA[i] 的每个二进制位一一对应。

死锁的检测算法如下:

(1) 任务 i 申请互斥资源 j 不成功,则令 mask = 1 < j 转 (2)。

(2) 检查资源分配矩阵,查找该占用该资源的任务。如果占用该资源的任务是 i,则形成环路,存在死锁。

```
for( k = 0; k < m; k++ ) {
    if( ( MatrixA[k] & mask ) != 0 ) {
        if( k == i ) { 存在死锁,转死锁解除程序; }
```

```
else { 转(3); }  
}  
}
```

(3) 检查资源请求矩阵,查找该任务请求并等待的资源,如果任务没有请求资源,则不存在死锁。

```
if( MatrixR[k] == 0) { 不存在死锁,算法结束; }  
else { mask = MatrixR[k]; 转(2); }
```

当检测到死锁时,就要采取措施将系统从死锁状态中解脱出来,可以用下面一些策略来解除死锁:①终止任务,通过一次性或者逐一撤消参与死锁的任务并回收它们所占用的全部资源。②剥夺资源,将死锁任务的全部或部分资源一次性或者逐步剥夺直到死锁解除为止。

由于死锁解除的方法均涉及损失了这些任务已经完成工作的开销,因此在基于成本的基础上选择逐步剥夺死锁环路中优先级最低任务所占用资源的方法,同优先级选择占用资源时间最短的任务,这样就需要在每个任务占用资源时增加一个计时器。

2.2 性能分析

使用资源请求、分配矩阵进行死锁检测的算法中,第(2)步循环查找占用资源的任务,查找成功的平均时间为 $O(m/2)$,最坏情况或查找失败为 $O(m)$,第(3)步查找任务请求并等待的资源,由于使用位操作,执行时间为常数 $O(1)$,外层循环次数取决于死锁环路中参与死锁的任务或资源数 h ,如果不存在死锁环路,则 h 为请求、分配链中任务或资源的数量 $h \leq \min(m, n)$,所需时间为 $O(m \times h)$ 。死锁解除算法中被剥夺资源的任务的选择可以在死锁检测的循环过程中查找,所需时间为常数 $O(1)$ 。因此死锁检测与解除的时间复杂度为 $O(m \times \min(m, n))$ 。

3 测试验证

为了对算法的有效性和实时性进行验证,把改进后的系统移植到 LPC2138 开发板上,编写测试用例进行测试。

例1 优先级继承协议测试 创建3个任务 Task1、Task2、Task3,优先级依次为 11、12、13,创建信号量 Mutex1,任务申请信号量的顺序依次为: Task3、Task1、Task2,运行结果如图2所示,证明改进的优先级继承协议可以很好地解决优先级反转问题。

任务名	优先级	动作
Task3	13	申请并得到Mutex1
Task1	11	申请Mutex1,阻塞
Task3	11	优先级提升
Task3	13	释放Mutex1,优先级还原
Task1	11	得到Mutex1
Task2	12	申请Mutex1,阻塞
Task1	11	释放Mutex1
Task2	12	得到Mutex1

图2 优先级继承协议测试结果

例2 死锁的检测与解除测试 死锁的检测时间与系统总任务数和参与死锁的任务或资源数量有关,并且检测结果也可能发现不存在死锁,即伪死锁。测试结果如表1所示。

表1 死锁检测算法测试结果

总任务数 m	参与的任务或资源数 h	死锁检测时间(μs)	伪死锁检测时间(μs)
20	4	17	14

总任务数 m	参与的任务或资源数 h	死锁检测时间(μs)	伪死锁检测时间(μs)
20	6	30	22
20	8	43	34
20	10	55	44
20	12	58	54
20	14	71	68
20	16	84	70
30	16	120	111
40	16	168	144
50	16	191	181

将数据分成两组进行分析。前7条数据为总任务数 m 相同,参与死锁的任务或资源数 h 不同,如图3所示,检测时间基本与 h 成正比。后4条数据为参与死锁的任务或资源数 h 相同,总任务数 m 不同,如图4所示,检测时间基本与 m 成正比。

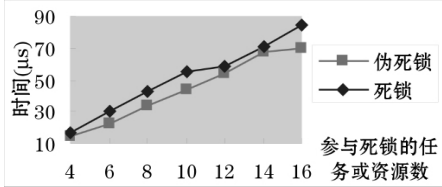


图3 总任务数相同时所用时间

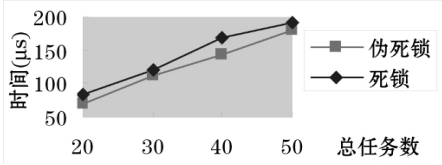


图4 参与死锁的任务或资源数相同时所用时间

通过以上测试与分析证明改进的算法有效可行,并且实时性能能够满足一般实时系统的需求。

4 结 语

本文通过对 $\mu C/OS-II$ 内核任务管理机制的改进,扩展了同优先级任务,从而可真正实现优先级继承协议解决优先级反转问题。在保证系统的实时性的前提下,将资源分配图用资源请求、分配矩阵来表示,在任务申请资源阻塞时进行死锁的检测与解除。系统改进后,在处理优先级反转和死锁问题时减少了程序员的直接参与。

参 考 文 献

[1] Jean J Labrosse. 嵌入式实时操作系统 $\mu C/OS-II$ [M]. 邵贝贝,译. 2版. 北京: 北京航空航天大学出版社, 2003.
[2] 沈金荣,刘翔. $\mu C/OS-II$ 内核结构分析及多任务调度实现[J]. 计算机工程, 2006, 32(23): 84-87.
[3] 吴永明,罗海据. $\mu C/OS-II$ 系统中任务调度与监控机制改进[J]. 计算机工程, 2009, 35(12): 266-268.
[4] 刘鹏,牛强,陈岱,等. 一种改进型优先级天花板协议设计与实现[J]. 计算机工程, 2008, 34(1): 66-68.
[5] 张琼声. 反馈 EDF 调度算法对 $C/OS-II$ 内核的改进与实现[J]. 计算机工程与设计, 2009, 29(16): 4128-4131.
[6] 周北海,王溪波,乔建忠,等. 基于多参数的 $\mu C/OS$ 任务优先级和调度方法[J]. 计算机工程, 2007, 33(21): 28-30.