支持 SRP 协议的实时调度技术①

马运南[1], 陈香兰[2]

[1](中国科学技术大学 软件学院, 合肥 230027)
[2](中国科学技术大学 计算机学院, 合肥 230027)

摘 要: 实时操作系统对多任务资源访问控制提出了较高的要求. 资源竞争常会引起优先级翻转问题导致任务阻塞, 增加资源等待延迟时间. 传统资源访问控制协议可以部分解决此问题, 但是存在上下文切换次数高, 任务调度效率低等不足. 在结合 SRP 协议的提前阻塞特性和 MiniCore 实时调度器设计后, 分析了协议中任务调度规则, 指出了调度器支持 SRP 协议时效率低下的原因, 并引入胜者树结构改进就绪队列, 给出了新的任务搜索算法. 理论分析与实验结果表明改进的调度器在任务集规模较大时, 提高了调度效率, 较好的支持了 SRP 协议.

关键词: 资源访问; 优先级翻转; SRP 协议; 胜者树; 时间开销

Real-Time Scheduling Technique Under SRP Protocol

MA Yun-Nan[1], CHEN Xiang-Lan[2]

[1](School of Software Engineering, University of Science and Technology of China, Hefei 230027, China)
[2](School of Computer Science, University of Science and Technology of China, Hefei 230027, China)

Abstract: Real-time operating systems lay claim to multitasking resource accessing control. Resources competition not only results in priority inversion and tasks blocking, but also prolongs timing delay caused by acquiring resources regularly. Although conventional resources accessing protocol works out priority inversion issues, there are still some defects such as frequent context switch, scheduling efficiency etc. With a research on the implement of MiniCore OS and SRP, this paper analyses the detailed criteria of scheduling and demonstrates the causation of low scheduling efficiency under SRP. To solve these issues above, we proposed and implemented a ready queue based on winner tree and a job selection mechanism with low timing overhead. Theoretical analyses and experiment results justify the better performance on large scale tasks scheduling under SRP protocol.

Key words: resources accessing; priority inversion; SRP protocol; winner tree; timing overhead

在实时系统中, 通常使用互斥量保证共享资源的安全访问, 但是互斥访问的信号量操作顺序不当可能会导致优先级翻转和死锁发生. 当死锁发生时任务会一直阻塞直到死锁检测机制强制释放任务保持的资源, 并使任务重新运行. 当多任务间优先级翻转发生时, 高优先级任务与低优先级任务竞争同一个资源而阻塞, 而且因阻塞产生的等待延迟没有时间上界. 为了保证实时系统中任务满足其自身的时间约束, 减少资源等待延迟, 实时系统必须提供共享资源的访问控制机制[1].

现阶段已有许多关于资源访问控制的研究. 典型的访问协议有优先级继承协议[2,3](Priority Inheritance Protocol, 简称 PIP)和天花板协议[2,3](Priority Ceiling Protocol, 简称 PCP)作为固定优先级系统下的资源访问控制方案. Chen 和 Lin 对 PCP 协议做了扩展, 提出了 PCP 协议的扩展动态优先级天花板协议[4](Dynamic Priority Ceiling Protocol, 简称 DPC)使其可以应用在动态调度算法中, 如 EDF 与 EDZL[8]调度. Jeffay 提出了动态时限修微调协议[5](Dynamci Deadline Modification,

① 基金项目:国家自然科学基金(61379040,61272131);江苏省自然科学基金(SBK2012194)
收稿时间:2015-05-08;收到修改稿时间:2015-06-08

简称 DDM), DDM 协议中系统为了避免不必要的抢占行为而微调竞争共享资源的任务时限.除 DPC 协议外, 还有 PIP 协议的扩展动态优先级继承协议[6](Dynamic Priority Inheritance Protocol, 简称 DPI). 在 DPC 协议中, 每当任务优先级改变时, 系统天花板和资源优先级天花板随之改变. Baker 提出了栈资源协议[7](Stack Resource Policy, 简称 SRP), SRP 协议可以同时在动态优先级与静态优先级系统中使用.

在使用 SRP 协议的系统中任务除拥有自身的优先级以外, 还有用于竞争资源的抢占优先级(preemption level), 这种独立的双优先级属性是 SRP 协议同时支持动态与静态优先级系统的关键. SRP 协议另一个特点是使用提前阻塞原则, 即任务在试图抢占其他运行中任务时进行资源分配测试, 如果没有通过测试则阻塞此任务. 而在 PCP 与 PIP 协议中, 任务在请求资源失败时被阻塞. 资源分配测试的目的是判断一个任务是否可能在执行后因访问共享资源而导致阻塞, 如果没有通过测试则调度器放弃此任务继续寻找下一个合适任务, 即使此任务是当前时限最小或者最高优先级者. 这种提前阻塞的方式, 保证了任务一旦开始执行就不会因为系统资源而阻塞, 降低了系统的并发性, 但是避免了不必要的上下文切换, 减少了系统开销.

协议中的资源分配测试过程在调度器选择最优任务时会消耗大量时间, 降低调度效率. 在最坏情况下, 即就绪队列中唯一的合适任务是时限最大者, 此时调度器必定是在对所有就绪任务都进行测试之后才找到最优任务执行. 在每个调度时机, 随着系统中任务集规模的增大, 调度器在资源分配测试过程中消耗的时间将明显增加, 降低了任务调度效率.

针对上述传统 RTOS 调度器在应用 SRP 协议上的不足, 本文首先分析了动态优先级系统在最优任务选择过程中的时间消耗, 指出了在传统 RTOS 调度器中任务选择的时间复杂度与任务集规模成正相关. 随后本文提出了一种调度器就绪队列改进设计和任务选择算法的改进, 在改进的调度器中最优任务选择的时间复杂度由 $O(n)$ 减少为 $O(\log_2 k)$.

本文的组织结构如下, 第一节给出了 SRP 协议规则与任务集调度实例, 第二节介绍现有实时系统 MiniCore 中调度器的设计并分析任务调度的时间开销, 第三节给出针对 SRP 协议的实时调度器改进, 第四节给出实验结果与分析, 第五节总结.

1 资源访问与任务调度开销

1.1 资源访问协议

在时限驱动的动态优先级系统中, 调度器根据任务的时限属性作出调度决断. Baker 根据这种时限驱动系统的特点扩展了在 RMS 调度器上使用的 PCP 协议. SRP 协议的提出依据了动态优先级系统中的资源竞争不会随着时间而变化这一现象, 动态系统中资源竞争关系可以同静态优先级系统一样做静态分析[3].

如图所示, 有三个任务 τ_1,τ_2,τ_3 依次到达,3 个任务优先级依次增加, τ_1 为最高优先权. 在 t = 0 时刻, τ_3 到达并开始执行, 在 t = 1 时刻, τ_3 请求临界资源 R 并将互斥量 M 上锁. 在 t = 2 时刻, τ_1 到达并抢占 τ_3 开始执行, 在 t = 3 时刻, τ_1 请求临界资源 R, 因为 R 已被 τ_3 独占所以 τ_1 被阻塞, 等待资源被释放. 在 t = 3 时刻, 因为 τ_1 阻塞所以处理器空闲, τ_3 恢复执行. 在 t = 4 时刻, τ_2 到达并抢占 τ_3, τ_1 的等待时间继续增加. 在 t = 5 时刻之后如果 τ_2 继续执行, 或者有与 τ_2 同优先级的任务到达, 那么 τ_1 会一直被阻塞直到超时. 这种高优先权任务 τ_1 因低优先权任务 τ_2 执行而等待的现象, 称为优先级翻转.

为了解决优先级翻转的问题, 在 SRP 协议中, 任务 τ_i 拥有除优先级和时限之外另一个影响任务间抢占的属性: 抢占优先级(preemption level) π_i. π_i 是在任务创建时赋予其的已知属性, 任务 τ_i 生成的所有服作业 j_i 将继承 π_i 作为其属性参数.

设任务 τ_i 的优先级表示为 ρ_i. 在使用 RMS 调度的系统中, τ_i 与作业 j_i 到达时间 r_i 和任务优先级 ρ_i 成函数关系, 三者间关系如定义 1 所示.

定义 1. 如果 $\rho_i > \rho_k$ 并且 $r_i > r_k$, 则 $\pi_i > \pi_k$ [2].

对定义 1 稍作修改, 将任务优先级替换成任务时限, 可得在三者在使用动态优先级调度的系统中的函数关系, 如定义 2 所示, 其中 D_i 是任务 τ_i 所产生作业 j_i 的相对时限.

定义 2. 如果 $D_i < D_k$, 则 $\pi_i > \pi_k$ [2].

在 SRP 协议中调度决断的判定依赖于任务(或作业)属性 π_i. 当系统为所有任务设置 π 后, 可以得出系统中资源优先级天花板 $\pi(R)$. 设 $\pi(R)$ 为当前系统中所有请求资源 R 的作业中, 抢占优先级最高的作业的 π 数值, 如式 1 所示. τ' 为请求资源 R 的任务子集.

$$\pi(R) = \max\{\pi_i \mid \tau \in \tau'\} \quad (1)$$

$$\Pi(t) = \max\{\pi(R)\} \quad (2)$$

在任意时刻 t, 系统优先级天花板 $\Pi(t)$ 设置为 t 时刻所有资源优先级天花板最大值, 如式 2 所示. 如果当前系统中所有资源都处于空闲状态, 那么 $\Pi(t)$ 设置为 Ω, 表示 $\Pi(t)$ 于当前所有任务的 π 值.

1.2 任务调度规则

1) 优先级天花板更新规则: 当系统中所有资源可用时, 系统优先级天花板 $\Pi(t)$ 设置为 Ω. $\Pi(t)$ 变更的时机为资源分配与释放的时刻;

2)任务调度规则: 当作业 j_i 被调度器选中执行时, 首先进行资源分配测试, 即判断任务抢占优先级是否大于系统优先级天花板 $\Pi(t)$ 且大于当前运行中作业 j_k 的抢占优先级 π_k, 即满足条件 $\pi_i > \Pi(t) \cap \pi_i > \pi_k$. 若是则作业执行, 若否则阻塞选择就绪队列中其他作业继续测试, 知道找到合适作业. 作业执行时根据其时限或者优先级以优先级驱动的方式调度执行.

3)资源分配规则: 在任意时刻, 如果有作业请求资源, 则直接分配资源给此作业.

在使用 SRP 协议时, 新作业可能因未通过资源分配测试被提前阻塞, 导致作业产生资源等待延迟时间, 延迟时间的上界由定理1给出. SRP协议中任务阻塞时机是任务间试图抢占的时刻, 而不是任务请求资源的时刻. 未通过资源分配测试的作业被提前阻塞, 继续在就绪队列中等待执行. 而当前运行中的作业均已通过测试, 测试保证了系统在此作业运行时可以满足其所有的资源需求, 不会因为资源请求而阻塞.

定理 1. 在使用 SRP 协议的系统中, 作业 j_i 的资源等待延迟时间至多为临界区代码的一次执行时间[6].

当系统使用 SRP 协议时, 任务集的可调度性判断需要考虑资源等待延迟时间, EDF 可调度性判定公式修改如式 3, EDZL 可调度性判定公式修改如式 4. 其中 B_k 是任务集产生的所有作业中等待延迟的最大值, C_i 与 T_i 分别为作业 j_i 的最大执行时间与周期性任务 τ_i 的周期.

$$\sum_{i=1}^{n}\frac{C_i}{T_i}+\frac{B_k}{T_k}\le 1 \tag{3}$$

$$\sum_{i=1}^{n}\frac{C_i}{T_i}+\frac{B_k}{T_k}\le m(1-\frac{1}{e}) \tag{4}$$

1.3 任务集调度实例

表 1 给出了在动态优先级系统中的任务集在一个超周期内作业的调度实例. 作业集合中 j_1 到 j_8 的相对时 D_i 依次增大, 抢占优先级 π_i 依次减少. 设系统初始 $\Pi(0)=0$, 资源 R_1,R_2,R_3 的优先级天花板为 $\pi(R_1)=6$, $\pi(R_2)=4$, $\pi(R_3)=5$, λ_i 表示作业 j_i 临界区代码执行时间长度.

表 1 任务集 π 在一个周期内的作业集合

j_i	r_i	D_i	π_i	C_i	λ_i	R_i
j_1	6	15	6	3	1	R_1
j_2	9	21	5	2	1	R_3
j_3	7	22	4	2	1	R_2
j_4	4	23	3	3	1	R_3
j_5	3	25	2	2	1	R_3
j_6	0	50	1	8	4	R_2

由表 1 可知, 在 $t=0$ 时刻, 作业 j_6 到达并开始执行. 在 $t=2$ 时刻, j_6 获得资源 R_2 进入临界区, 此时系统天花板 $\Pi(2)=\pi(R_2)=4$. 在时刻 $t=6$ 之前, j_6 还处于临界区中, j_4,j_5 分别到达, 虽然此时 j_4,j_5 的时限比 j_6 小, 但是因为 j_4,j_5 的抢占优先级小于当前系统天花板, 即 $\pi_{4,5}\le\Pi(6)$, 所以作业 j_4,j_5 被提前阻塞. 同时在 $t=6$ 时刻, 作业 j_1 到达, 又因 $\pi_1>\Pi(6)$ 所以 j_1 立即抢占 j_6 获得处理器. 在 $t=8$ 时刻, j_1 进入临界区, 获得资源 R_1, 此时系统天花板为 $\Pi(8)=6$. 在此之前, j_3 到达, 因 $\pi_3<\Pi(8)$ 所以被提前阻塞. 经过 1 个时间单位在 $t=9$ 时, j_1 执行完毕, 此时系统天花板降回 $\Pi(9)=4$, 此时调度器需从就绪队列的作业子集 $\{j_2,j_3,j_4,j_5,j_6\}$ 中选择满足条件 $\pi_i>\Pi(9)$ 且时限 d_i 最小者为最优任务, 最终选中作业 j_2 执行. 任务集 π 的执行实例如图 1 所示.

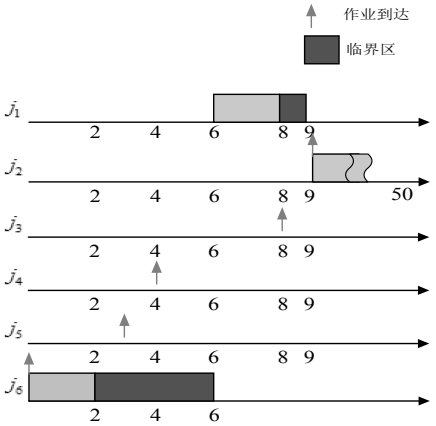

图 1 基于 SRP 任务集 τ 调度实例

任务集 π 在不使用 SRP 协议时的调度实例如图 2 所示, 可以看出作业 j_3 与作业 j_6 竞争资源 R_2, 且因为 j_6 保持资源时被抢占处理器, 导致 j_3 产生的资源等待延迟为 22 个时间单位.

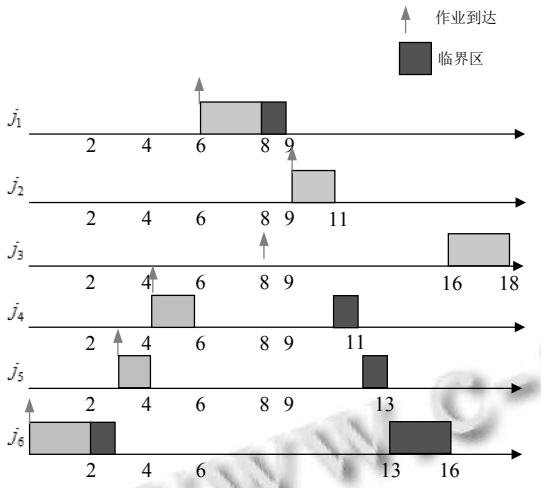

图 2　任务集 τ 调度实例

2　MiniCore实时调度器

MiniCore[8,9]是基于 SEFM 操作系统模型[8,9]的嵌入式实时操作系统, 供了任务的管理与调度、任务间通信与同步、内存管理、实时时钟管理、中断服务等基本功能. MiniCore 中使用任务-服务模型作为基本的运行抽象. 任务由若干个服务构成, 这些服务可以使串行的, 也可以是并行的, 服务间不可嵌套. 系统中的服务于任务是多对多的关系, 一个服务可以使多个任务的组件, 一个任务也可由多个服务组合而成. 任务的运行实例是服务组合的运行实例, 即服务实例小端口的组合. 为简化问题描述, 本文中 MiniCore 的服务的调度等价为任务调度, 服务实例等价为任务作业.

2.1 MiniCore 就绪队列

MiniCore 中就绪队列使用平衡树与链表作为基本结构. 在多数 RTOS 中, 每隔几毫秒就会有一个新的系统事件发生如作业完成执行、新作业到达等, 这就要求平衡树在进行频繁的维护操作时不需要每次都执行平衡操作. 因此从维护效率方面考虑, 就绪队列使用 RBT 结构挂载作业. 基于 2-3-4 树的 RBT 可以不用每次作业到达就执行平衡操作, 仅需要直接插入作业, 而且优化的 RBT 如 LLRBT[10]通过限定红链接只能向左倾斜从而避免了大量调整右倾红链接的平衡操做, 缩短了维护时间.

当新作业到达被插入就绪队列时, 调度器以 $O(\log_2 n)$ 的复杂度根据其时限 d_i 放入 RBT 中合适的位置, 即树中前驱节点的时限小于新作业, 后续节点的时限大于新作业. 在插入过程中, 伴随着前驱与后继节点的查找, 所以再插入新作业时, 同时将其与前驱节点后继节点使用双链表连接起来. 链表的作用是为了快速找到拥有最早时限 d_i 的作业节点, 时间复杂度为 $O(1)$, 而通过 RBT 树查找此节点则需要搜索树中所有左侧子树, 显然时间复杂度为 $O(\log_2 n)$ 远大于前者. 就绪队列的删除操作与 RBT 相同.

这种 RBT 树和链表结合的方式, 使得内核不必时时维护链表中作业的顺序. 任务控制块中的前驱后继指针使内核不需要在树形作业队列之外维护一个线性表存储存作业有序执行顺序的排列, 降低了空间开销, 并简化了插入、删除作业等维护操作, 加快了作业搜索操作. 根据上述操作可知, 任务控制块结构中含如下关键数据成员.

表 2　任务控制块部分数据成员

```
typedef enum{ RED, BLACK } Node_Color;
typedef Uint32 Deadline_Ctrl;
typedef struct miniport_ctrl {
        Deadline_Ctrl absolute_deadline;
        Deadline_Ctrl relative_deadline;
        Pminiport_Ctrl * left , * right , * parent;
        Double_Link task_link;
        Node_Color color;
} Miniport_Ctrl, * Pminiport_Ctrl;
```

任务控制块中的 Pminiport _Ctrl 是其队列组织必需的 RBT 树数据成员, 作用是构建一个 LL-RBT 树形队列. 就绪队列有一个头结点 Miniport _Chain _Ctrl 用以指向队列的树根部, 拥有 2 个 Pminiport _Ctrl 成员, 一个是 root 指向树根, 一个是 first 指向队列中时限最小的任务控制块结点.

表 3　队列辅助头结点

```
typedef struct {
        Pminiport_Ctrl root;
        Pminiport_Ctrl first;
} Miniport_Chain_Ctrl;
Miniport_Chain_Ctrl EDF_queue, EDZL_queue;
```

就绪队列中作业的插入和删除维护操作算法如表 4, 表 5 所示. 因使用 EDF 和 EDZL[11]算法, 所以 MiniCore 最优作业搜索算法仅需直接通过

Miniport_Chain _Ctrl.first 直接定位到 RBT 最左节点即可, 或者通过 Miniport _Chain _Ctrl.root 自顶向下遍历到最左节点.

表 4 作业插入算法

```
if parent is not NULL then
        if new Miniport's di < parent's di then
                new Miniport is parent's left child
                if parent is first in queue then
                        new Miniport is first
                end if
        else
                new Miniport is parent's right child
        end if
else if queue is EMPTY then
        new Miniport is the root and is the first
end if
```

表 5 作业删除算法

```
if Miniport is first in queue and first Miniport must be updated
then
        if first Miniport has right child then
                Miniport's right child is first
        else
                first Miniport's parent Miniport is first
        end if
end if
```

2.2 时间开销分析

SRP 协议的抢占优先级以及提前阻塞设计虽然避免了任务因请求资源而阻塞, 但是增大了调度器选择最优任务执行的开销. 在就绪队列查找合适作业的过程中, 必须进行两次比较: 第一次比较找出时限最小或者优先级最高的作业, 第二次比较已找到作业的抢占优先级是否高于系统优先级天花板. 在使用 EDF 或者 EDZL 等算法的 MiniCore 系统中, 如果使用 SRP 协议设计控制其资源访问, 则调度器选中的任务可能并不是最优任务. 例如一个作业的时限是就绪队列中最小的, 但是其抢占优先级 $\pi_i \le \Pi(t)$, 那么为了避免阻塞此作业会被放弃调度执行. 在每个调度时刻, 调度器总是选择较小时限的作业子集中满足条件 $\pi_i > \Pi(t)$ 的最小时限作业. 在最坏情况下, 调度器必须对就绪队列中所有作业都进行抢占测试, 这就造成了巨大的时间开销. 如表 1 中所示的任务集, 在 $t=9$ 时, 调度器需要逐一搜索整个就绪队列找到将要执行的作业.

$t=9$ 时在就绪队列上剩余 7 个作业以绝对时限 d_i 排序, 其组织方式如所图 2 所示. 剩余的作业子集 $J'=\{j_2,j_3,j_4,j_5,j_6\}$ 存储在 LLRBT 上, 同时一个双链连接所有的作业.

由图 3 可知, 在 $t=9$ 时刻, 虽然以 $O(1)$ 时间复杂度找到了时限最小的作业 j_4. 但是因为当前 $\Pi(9)=4$, 而 $\pi_4=3$, 所以 j_4 不满足可调度条件, 需要继续向后查找. 又所有任务在双链上是按照 d_i 递增有序的, 所以调度器从 j_4 进行抢占测试直到 j_2 共经过 4 个节点, 最终得到满足条件 $\tau_2 > \Pi(9)$ 的作业 j_2.

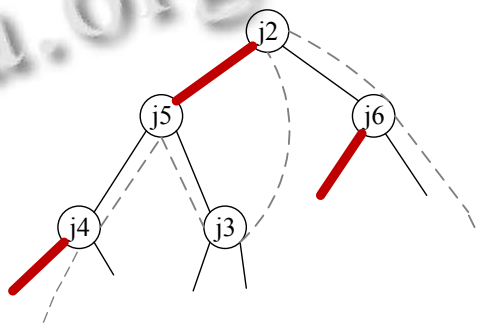

图 3 就绪队列中作业子集 J'

因此在 $t=9$ 时, 调度器选择 j_2 执行, j_6 继续等待. 分析可知, 在最坏情况下调度器可能需要测试就绪队列上所有的作业才能得到最优任务, 此时的时间复杂度为 $O(n)$, n 为任务集规模. 查找合适作业所进行的测试需要大量的时间开销, 这些开销随着作业数量增大而增加. 为了减少调度器选择任务过程的时间复杂度, 需要在原有就绪队列的基础上考虑任务抢占优先级属性加以改进.

3 调度器基于SRP协议的优化

3.1 优化需求分析

在时限驱动系统中, 调度器需要将任务的抢占优先级和作业时限属性同时作为调度参数, 在就绪队列中选择满足 $\pi_i > \Pi(t)$ 且时限最小的作业. 为了高效选择适合作业执行, 就绪队列需要按照抢占优先级属性为第一要素重新设计, 并且需要考虑调度器对作业时限做大小比较的时间复杂度. 因此降低寻找下一个最小元素所需要进行的比较次数最少的结构是设计就绪队列的基本需求.

设队列结构中的节点作为任务控制块 Miniport _Ctrl 结构. 为了最小化查找最优作业的时间开销, 就

绪队列的维护与搜索机制需要同时考虑作业时限 d_i 与任务抢占优先级 π_i 两种属性. 在使用 SRP 协议的 MiniCore 系统中, 调度器应当首先搜索抢占优先级高于系统天花板的作业子集 $J_{\pi>\Pi}$, 然后在集合 $J_{\pi>\Pi}$ 中搜索时限最小的作业为最优任务赋予处理器运行.

3.2 优化方案

最优作业搜索可以抽象为作业的一次预选与作业的最优解选择过程, 两个处理过程分别基于不同的任务属性. 作业预选的目的是从所有候选者中选取按照 π_i 排序前K大的作业, 选择最优解的目的是从 π_i 前K大的作业中选中 d_i 最小的作业.

为了高效选取前 K 大作业, 就绪队列需要组织成堆结构. 堆结构的特点是能以 $O(1)$ 的复杂度选取前K大元素和极值, 加入新的元素后以 $O(\log_2 k)$ 的复杂度修复堆即可. 然而原始的堆结构在更新修复时, 需要比较三个节点: 一个父节点和两个子节点. 本文选择堆结构的一种优化变种: 自底向上调整的胜者树作为就绪队列基本结构. 胜者树每层节点中的调整过程仅需要与兄弟节点比较一次即可, 相对于原始堆结构调整过程比较次数更少, 时间更快.

胜者树的特点是中间结点记录的是胜者的标记, 胜者树将参与选择的作业作为叶子节点, 非叶子节点保存叶子间的竞争信息, 根节点保存最终获胜的作业信息. 如果一个作业的时限发生改变, 那么修改此胜者树也很容易. 只需沿着从该做而已节点到树根节点的路径修改胜者树, 不用改变其他作业的节点记录.

3.3 作业组织方式

为了最小化查找最优作业的时间开销, 需要按照任务抢占优先级为参数将作业挂载到一棵搜索树上. 经上节分析知此搜索树为胜者树, 树中节点分为两类: 叶子结点和非叶子节点. 叶子结点代表一个作业, 叶子的 key 是任务当前作业的时限, 每个叶子节点都与一个唯一的抢占优先级关联; 非叶子节点是胜者树中的 2 棵子树根的胜者记录, 节点的 key 是胜者作业的时限. 所有叶子从左到右以任务抢占优先级增序排列. 因为胜者树作为基本结构, 所以任务搜索树是完全二叉树, 最底层的叶子即作业数量为 $2\lceil \log_2 k \rceil$, 树高为 $\lceil \log_2 k \rceil$, k 为系统中不同 π 数值的数量. 为了简化问题, 假设系统中每个作业的 π 互不相同. 如果当前抢占优先级 π 上作业为空, 则对应的叶子节点 $key=\infty$. 为了描述的统一, 依然给此 π 上挂载一个虚拟作业, 此虚拟作业是不存在的, $key=\infty$ 表示此虚拟作业的时限为无限大. 树中的非叶子节点记录着节点 2 棵子树的时限最小值, 同理树的根结点是整个作业集合中的时限最小值.

当系统中没有任务时, 非叶子节点为空集. 当有新作业到达时, 根据其 π_i 选择对应的叶子结点填充时限数值, 并将叶子与 Miniport _Ctrl 结构关联. 当新作业陆续到达后对应每个叶子结点, 每个非叶子的生成过程相当于一次竞争, 每一次竞争选择非叶子节点 2 个子树根中 key 最小值. 非叶子节点从倒数第二层一直生成, 直到整棵树的根部. 当作业执行完毕后, 对应叶子节点为空, 即 $key=\infty$.

图 3 给出了当系统天花板为 $\Pi(t)=0$ 时, 作业集合 $J=\{j_1,j_2,j_3,j_4,j_5,j_6\}$ 在就绪队列中的组织结构, 其中 $d_1=10,\pi_1=1$; $d_2=25,\pi_2=4$; $d_3=15,\pi_3=5$; $d_4=60,\pi_4=6$; $d_5=20,\pi_5=7$; $d_6=23,\pi_6=8$.

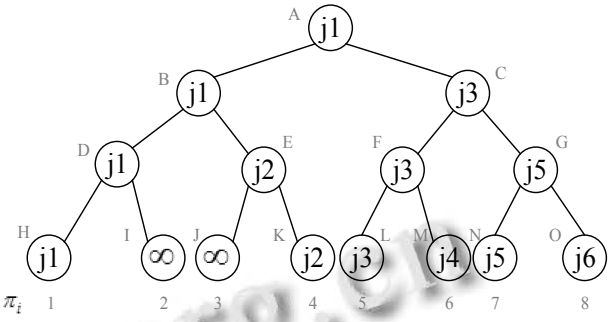

图 4　胜者树中的作业集 J

3.4 任务搜索

根据上节任务集的就绪队列结构, 可得最优作业(或任务)搜索算法如表 5 所示, 系统天花板表示为 $\Pi(t)=S$. 以图 3 所示作业集合为例说明最优作业选择过程, 设当前系统 $\Pi(t)=2$.

第一轮搜索. candidate = J, subroot = J; subroot is left child, competitor = K, 又有 $D_k < D_j$, candidate = K; subroot = E; E ≠ ROOT, goto revalidation.

第二轮搜索. subroot is right child; subroot = B; B ≠ ROOT; goto revalidation.

第三轮搜索. subroot is left child, competitor = C, 又有 $D_c < D_k$, candidate = C; subroot = A; A = ROOT; return C = (j_3).

表 5 搜索满足 $\pi_i > S$ 的最优作业

```
    candidate=leaf node which ;
    subroot=candidate;
revalidation:
    if subroot=left child then
        competitor=sibling of subroot;
        if then
            candidate=competitor;
        end if
    end if
    subroot=parent of subroot;
    if subroot=ROOT of the whole tree then
        return candidate;
    end if
    goto revalidation;
```

4 实验分析

4.1 实验设计与统计结果

本节设计一组基于任务集的调度实验来测试文中调度器优化前后的系统开销. 任务的规模可变, 由 10 增加至 100 个任务. 目标系统 MiniCore 运行在 QEMU 的 i386 单核模拟器上, 测试例程通过调用 rdtsc 指令获得任务搜索过程开始结束时的处理器周期数之差, 进而根据常量 CPU 主频计算出每个任务集最优作业搜索过程消耗的时间大小.

本节设计一组基于任务集 τ 的调度实验来测试文中调度器优化前后的系统开销. 任务 τ 的规模可变, 由 10 增加至 100 个任务. 目标系统 MiniCore 运行在 QEMU 的 i386 单核模拟器上, 测试例程通过调用 rdtsc 指令获得任务搜索过程开始结束时的处理器周期数之差, 进而根据常量 CPU 主频计算出每个任务集最优作业搜索过程消耗的时间大小.

如图 5 所示, 测试分别比较了相同任务集在最坏情况下的搜索时间开销. X 轴表示时间单位微秒, Y 轴表示任务集规模. 此时作业集合中仅有一个作业抢占优先级高于系统天花板 $\pi_i > \Pi(t)$, 且此作业的绝对时限是作业集合中的最大值. 从图中可以得出胜者树与 LLRBT 的时间开销增长趋势分别为对数增长与线性增长, 本文的优化方法可以节约大量时间.

图 6 描述了在任务集 τ 在上述测试中所需要的空间大小, X 轴表示节点数, Y 轴为任务集规模. 统计量不包括作业的堆栈和就绪队列中辅助结构等, 仅统计挂载作业集合的LLRBT与胜者树结构中节点数目. 可以看出本文的优化方法需要较大的额外空间, 在任务集规模为 100 时, 胜者树需要 2 倍于 LLRBT 的空间. 这是因为胜者树需要额外的节点记录竞争胜出信息.

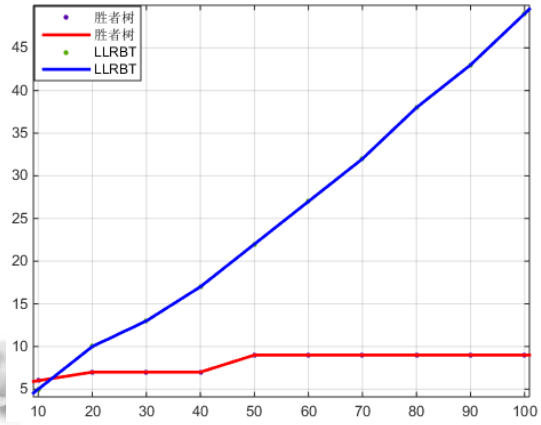

图 5 胜者树与 LLRBT 时间开销

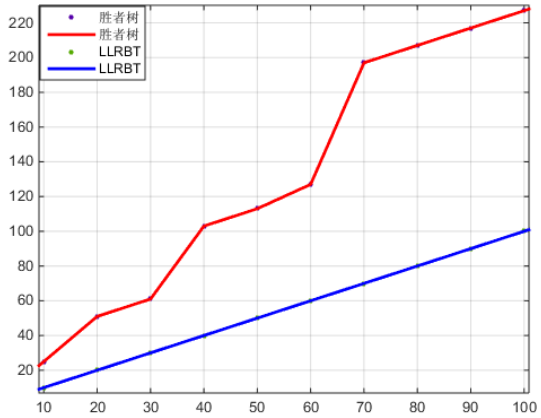

图 6 胜者树与 LLRBT 空间开销

4.2 结果分析

作业搜索效率取决于当前系统优先级天花板 $\Pi(t)$, 当 $\Pi(t)$ 非常大时, 在第一步的候选节点数量将会非常少, 那么后续查找合适任务的时间消耗就会很小. 上节算法的搜索过程中, 总共需要 3 ($S_{\max}$ - S - 1) + 2 次赋值操作, 2 ($S_{\max}$ - S - 1) + 1 次比较操作, 其中 $S_{\max}$ 为系统中任务抢占优先级 π_i 最大值.

在描述就绪队列优化需求时, 为了简化问题, 假设了系统中每个任务的 π_i 互不相同. 而在实际应用中, 在同一 π_i 数值上, 经常有多个作业与之关联. 所以就绪队列结构需要稍作修改使用 K 路胜者树, 在胜者树搜索结构的 K 个叶子节点后均挂载一个 FIFO 队列, FIFO 队列中的作业其 π 与叶子节点代表的作业 π_i 数值相同.

上述改进节省了搜索时间和避免了就绪队列平衡

性维护操作, 提高了调度效率, 同时也带来了空间上的开销. 队列结构中需要非叶子节点作为保存作业间竞争胜出的信息节点, 并不储存作业信息. 对于规模为n的任务集, 共需要$n-1$个额外的信息节点. 如果系统中主存资源不是非常小, 那么此空间开销是可以接受的, 因为与原调度器中线性的时间开销相比, 少量空间消耗带来调度效率的提升更为重要.

5 总结

本文介绍了实时操作系统中用于共享资源访问控制的 SRP 协议, 讨论了 SRP 协议中具体的任务调度规则, 并以任务集调度实例说明 SRP 协议的任务提前阻塞与双优先级属性特点, 这两种特性保证了任务一旦开始执行就不会因为系统资源而阻塞, 避免了因等待资源而产生的上下文切换. 同时还分析了 MiniCore 系统实时调度器在应用 SRP 协议上的不足之处, 即基于 EDF 和 EDZL 等动态优先级算法的就绪队列和作业搜索机制不能支持 SRP 协议中作业搜索第一阶段的作业预选需求. 在最坏情况下, 调度器对所有作业进行测试, 任务调度的时间复杂度与任务集规模成线性关系.

随后本文引入了实时调度器的改进方案, 使用 K 路胜者树替换原调度器中的 LLRBT 结构作为就绪队列, 给出新的作业搜索算法, 并讨论了改进后的作业搜索时间开销. 分析表明当使用 SRP 协议的系统中任务集规模较大时, 基于 K 路胜者树的改进调度器的优势在于任务调度效率方面. 下一步的研究工作将在保持调度效率的同时减少空间消耗方面展开.

参考文献

1 Hahn S, Reineke J, Wilhelm R. Towards compositionality in execution time analysis-definition and challenges. 6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems. 2013.

2 Buttazzo GC. Hard real-time computing systems: predictable scheduling algorithms and applications. Springer Science & Business Media, 2011.

3 Liu F, Narayanan A, Bai Q. Real-time systems. 2000.

4 Chen MI, Lin KJ. Dynamic priority ceilings: A concurrency control protocol for real-time systems. Real-Time Systems, 1990, 2(4): 325–346.

5 Jeffay K. Scheduling sporadic tasks with shared resources in hard-real-time systems. Real-Time Systems Symposium, 1992. IEEE, 1992: 89–99.

6 Buttazzo GC, Stankovic JA. Adding robustness in dynamic preemptive scheduling. Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems. Springer US, 1995: 67–88.

7 Baker TP. Stack-based scheduling of realtime processes. Real-Time Systems, 1991, 3(1): 67–99.

8 龚育昌,张晔,李曦,等.一种新型的构件化操作系统的内核设计.小型微型计算机系统,2009,30(1):1–7.

9 龚育昌,陈香兰,李曦,等.基于服务体/执行流模型的操作系统,2007.

10 Robert S. Left-Leaning Red-Black Trees. http://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf

11 Chao YH, Lin SS, Lin KJ. Schedulability issues for EDZL scheduling on real-time multiprocessor systems. Information Processing Letters, 2008, 107(5): 158–164.