

$\mu\text{C}/\text{OS}-\text{II}$ 中消息队列通信的数据安全问题

曾蜀芳¹, 郭 兵¹, 沈 艳²

(1. 四川大学 计算机学院, 四川 成都 610064;

2. 电子科技大学 机械电子工程学院, 四川 成都 610054)

摘 要: $\mu\text{C}/\text{OS}-\text{II}$ 是一个可移植、可裁剪的占先式多任务实时内核, 且源码开放, 便于学习、移植和维护。消息队列是一种广泛使用且灵活的任务间通信机制。分析了 $\mu\text{C}/\text{OS}-\text{II}$ 的消息队列通信机制中存在的数据库安全性问题, 然后通过改进消息队列通信所涉及的数据结构, 同时增加消息队列测试函数 $\text{OSQTest}()$, 以提供应用程序对消息队列中消息访问是否安全的测试途径, 从而提高了应用程序间通过消息队列进行通信的数据库安全性。通过模拟实验证明此改进在实际应用中的有效性。

关键词: $\mu\text{C}/\text{OS}-\text{II}$; 实时操作系统; 消息队列; 数据库安全

中图分类号: TP316.2

文献标识码: A

文章编号: 1673-629X(2009)08-0151-04

Data Security of Message Queue Communication in $\mu\text{C}/\text{OS}-\text{II}$

ZENG Shu-fang¹, GUO Bing¹, SHEN Yan²

(1. Department of Computer Science, Sichuan University, Chengdu 610064, China;

2. Department of Electronic Science and Technology, University of Electronic Science and Technology, Chengdu 610054, China)

Abstract: $\mu\text{C}/\text{OS}-\text{II}$ is a transplantable and cuttable multiple-task preemptive OS kernel which has opened its code. It is easy to study and transplant and easy for maintenance. Message queue is a widely used interprocess communication mechanism. First, analyse the data protection problem in $\mu\text{C}/\text{OS}-\text{II}$'s message queue mechanism, then improve the data structure referred and add message queue test function $\text{OSQTest}()$ which can be used to check if the access of a message in message queue is safe. All of these work can enhance the security of data which is exchanged between application processes. And at last prove this improvement is effective by simulative experiment.

Key words: $\mu\text{C}/\text{OS}-\text{II}$; RTOS; message queue; data security

0 引言

在多任务的实时系统中, 一项工作的完成往往需要通过多个任务或多个任务与多个中断处理程序共同完成^[1], 因此任务间的通信和同步是必不可少的, 可移植、可固化、可裁剪的占先式多任务实时内核 $\mu\text{C}/\text{OS}-\text{II}$ ^[2] 提供了五种通信和同步机制: 信号量、互斥信号量、消息邮箱、消息队列和事件标志组。其中消息队列允许一个任务或中断服务子程序向另一个任务发送以指针方式定义的变量或其他任务。因具体应用的不同, 每个指针指向的包含了消息的数据结构的变量类型也有所不同^[3], 因此消息队列是一种灵活的通信机制, 但它在给应用带来灵活性的同时也产生了数据库安全的问题。

文中通过对消息队列的数据结构和对消息队列的操作进行改进来提高通信中的数据库安全性。

1 $\mu\text{C}/\text{OS}-\text{II}$ 中消息队列通信机制

1.1 实现原理

$\mu\text{C}/\text{OS}-\text{II}$ 一共提供了 9 个对消息队列进行操作的函数: $\text{OSQCreate}()$ 、 $\text{OSQPost}()$ 、 $\text{OSQPend}()$ 、 $\text{OSQAccept}()$ 、 $\text{OSQPostFront}()$ 、 $\text{OSQPostOpt}()$ 、 $\text{OSQFlush}()$ 、 $\text{OSQDel}()$ 、 $\text{OSQQuery}()$, 其中前面 4 个为内核提供的典型服务, 分别可描述为:

- 消息队列初始化, 队列初始化时总是清空;
- 放一则消息到队列中去;
- 等待一则消息的到来;
- 如果队列中有消息则任务得到消息, 如果此时队列为空内核并不将该任务挂起, 而是用特别的消息代

收稿日期: 2008-11-17; 修回日期: 2009-02-06

基金项目: 国家科技部 863 计划项目 (2008AA01Z105)

作者简介: 曾蜀芳 (1983-), 女, 四川达州人, 硕士研究生, 研究方向为嵌入式实时系统; 郭 兵, 副教授, 研究方向为嵌入式实时系统、soc 和中间件。

码通知调用者^[4]。

上述 9 个函数中的 OSQCreate()、OSQDel()、OSQPend()、OSQQuery() 只有任务才能调用,而其他任务或中断服务程序都可以调用,同时除了 OSQCreate() 和 OSQPend() 不能单独禁用外,其他的都可以通过配置 OS_CFG.H 中的相应位来禁用或启用。

前述 9 个函数通过操作以消息队列所需的各种数据结构来实现消息队列的通信,这些数据结构及相互间的关系如图 1 所示。

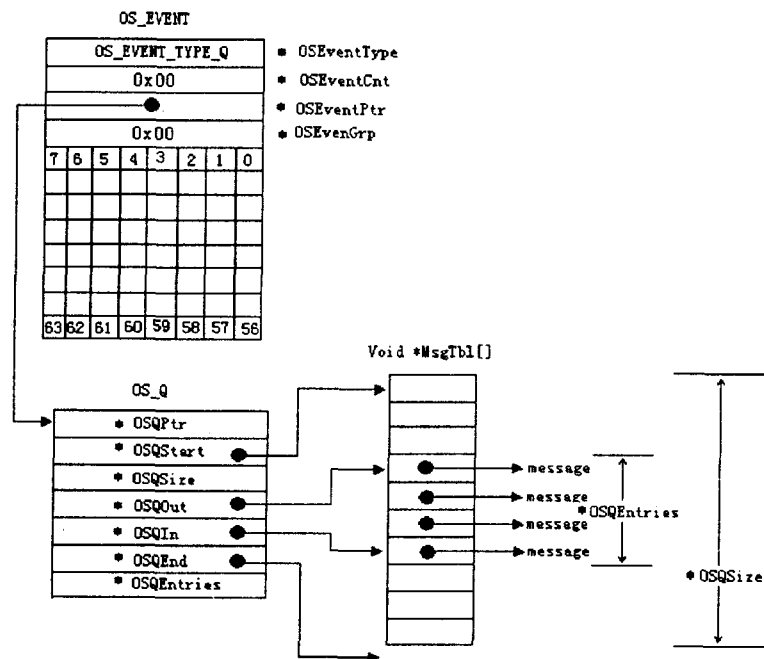


图 1 用于消息队列通信的数据结构

$\mu\text{C}/\text{OS-II}$ 中各种通信和同步的实现都需要一个 OS_EVENT 类型的事件时间控制块为基础,并根据所需实现机制的不同对事件控制块中各个参数进行不同的设置,对于消息队列通信来说在调用 OSQCreate 函数创建一个消息队列时, $\mu\text{C}/\text{OS-II}$ 会向系统申请一个事件控制块并对控制块中各项元素做如下设置:OSEventType 设为 OS_EVENT_TYPE_Q, OSEventCnt 设为 0, OSEventPtr 则设为一个指向队列控制块的指针。队列控制块是一个用于维护消息队列信息的数据结构,包含的元素如图 1 所示,其中 OSQSize 为消息队列可以容纳的最大消息数, OSQEntries 为消息队列中当前可用的消息数,而 OSQStart、OSQOut、OSQIn 为指向用户数据的指针型元素,每当接收到一个消息时 OSQIn 向下移动一个单位且 OSQEntries 加 1,发出一个消息时 OSQOut 向下移动一个单位且 OSQEntries 减 1,应特别注意的是 OSQEnd 指向消息队列结束单元的下一个地址的指针,并通过在 OSQPost 和 OSQPend 中对 OSQIn 和 OSQOut 的检测和修改来使得对

消息队列的操作不超出消息数组而构成一个循环的缓冲区。

1.2 $\mu\text{C}/\text{OS-II}$ 消息队列中的数据安全性问题

消息队列给任务提供了一种异步通信方式^[5],大大降低了程序间的耦合度,应用程序只需要发出消息就可以继续做其它事情,消息传输的发生不会强制应用程序等待发送结束;接收方也无需监视接收的全过程,而只要简单地从队列中读取消息并处理^[6],但在降低程序间耦合度的同时,无法保证最终的接收任务一

定能安全地接收到数据^[7]。由上一节的叙述可知,消息队列的通信通过操作 OS_Q 中的 OSQOut 和 OSQIn 等元素实现,而 OSQOut 和 OSQIn 为指向指针的指针,其所指向的指针在用户的应用程序和操作系统内核之间传递,从而这个指针所指向内容可能在操作系统内核毫不知情的情况下被应用程序改变,而在目前的 $\mu\text{C}/\text{OS-II}$ 中只能由应用程序员来对这种情况进行处理,即 $\mu\text{C}/\text{OS-II}$ 本身没有提供数据的保护。

以下为一个数据安全问题的例子:

对于消息队列 Q,当前有两个应用任务 T1 和 T2 分别向其发送和接收消息,初始时 OSQIn = OSQStart, OSQOut = OSQStart,在 t1 时刻任务 T1 向 Q 发送一个内容为“message1”地址为 0x8000 的消息,从而 OSQIn = OSQStart + 1,而 * OSQIn = 0x8000。按计划在 t2 时刻 T1 再次发送消息内容为“message2”的消息,且 T2 应该在 t1 和 t2 之间从消息队列中取出消息,但是若因为某一事件的发生使得 T2 延迟了取操作,又因为嵌入式系统中存储资源一般比较紧张^[8],T1 不可能每次发送消息时都重新申请地址,因此 T1 往往用同一地址,此处为 0x8000 来发送自己的消息,那么在 t2 时刻 T1 发送的“message2”就可能覆盖“message1”,致使“message1”丢失,具体如图 2 所示。

2 对 $\mu\text{C}/\text{OS-II}$ 消息队列的改进

要消除 1.2 中所述的数据丢失问题就必须给应用程序提供一个查询修改当前消息是否安全的途径,以使得当应用逻辑不确定在当前修改消息是否可靠时可以通过查询来确定。虽然 $\mu\text{C}/\text{OS-II}$ 提供了一个获取消息队列状态的函数 OSQQuery() 函数,但这个函数并不能向应用程序提供与调用程序相关联消息的使用状况,因此增加一个查询函数 OSQTest (OS_EVENT

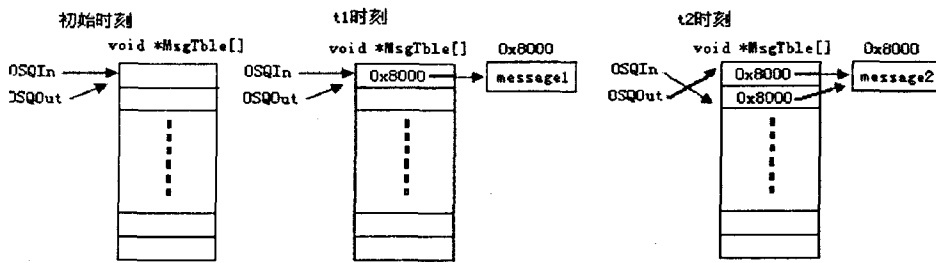


图2 存在数据安全问题的例子

$*pevent, INT8U *msgId, INT8U *circleNum$), 其中 $pevent$ 为此项处理涉及的消息队列事件控制块, $msgId$ 为调用任务前一次发送消息的消息在消息数组 $MsgTbl$ 中的位置, $circleNum$ 为当前消息队列的轮数, 其用途会在后面介绍。为了对这个函数提供支持还必须对用于消息队列的数据结构进行相应的改进, 包括: 在 OS_Q 结构体中增加元素 $INT8U *OSQFlagPtr$ 和 $INT8U OSQCircleNum$ 。其中 $OSQFlagPtr$ 指向一个长度为 $OSQSize$ 的数组, 其中每一位对应消息数组 $MsgTbl$ 相应位的信息是否已经被处理, 1 表示已经被处理, 即修改这个指针指向的内容是安全的, 否则不安全, 这个数组的内容由 $OSQPost$ 和 $OSQPend$ 函数在 $OSQCircleNum$ 的基础上进行处理, $OSQPend$ 中相关处理为:

```
void *OSQPend(OS_EVENT *pevent, INT8U *err)
{
    .....
    pq = (OS_Q *)pevent -> OSEventPtr;
    if(pq -> OSQEntries > 0)
    {
        .....
        num = pq -> OSQOut - pq -> OSQStart;
        pq -> OSQFlagPtr[num] = (INT8U)0;
        .....
    }
    .....
}
```

$OSQPost$ 中的相关处理为:

```
INT8U OSQPost(OS_EVENT *pevent, void *msg, INT8U *
circleNum, INT8U *msgId)
{
    .....
    pq -> OSQFlagPtr[*msgId] = (INT8U)1;
    .....
}
```

$OSQCircleNum$ 的设定是出于这样一种考虑, 即某一任务 $T1$ 在一次发送消息时获知自己发送的消息存放在消息数组的第 k 位, 一段时间过后当 $T1$ 需检测自己上次发送的消息是否已被使用时不能简单检测

$OSQOut$ 是否先于 k , 因为可能 $OSQOut$ 不先于 k , 但自己的消息已被使用, 这是由于 $OSQOut$ 已经进入下一轮, 当前的 k 处存放的已经是另一个任务 $T2$ 发送来的消息。

每当 $OSQOut$ 进入下一轮时 $OSQCircleNum$ 加 1 并对 256 取模, 可见若在极端情况下, 某一任务在发送完一条消息后刚好等待消息队列的 $OSQOut$ 轮转了 256 次后再次发送消息则可能会在检测时获得更改数据不安全的信号, 但这种极端情况发生的几率很小, 且即便发生也只需等待到 $OSQOut$ 进入下一轮后检测就能得到正确的结果。以下。为 $OSQTest$ 函数源码:

```
int OSQTest(OS_EVENT *pevent, INT8U *circleNum)
{
    OS_Q *pq;
    pthread_mutex_lock(&q_mutex);
    pq = (OS_Q *)pevent -> OSEventPtr;
    if(pq -> OSQFlagPtr[*msgId] == (INT8U)0)
    {
        pthread_mutex_unlock(&q_mutex);
        return 1;
    }
    else if(*circleNum != pq -> OSQCircleNum)
    {
        pthread_mutex_unlock(&q_mutex);
        return 1;
    }
    else
    {
        pthread_mutex_unlock(&q_mutex);
        return 0;
    }
}
```

由于对消息队列的数据结构进行了改进, 因此在创建消息队列和进行请求消息和发送消息时要求应用程序提供的参数也有所不同, 对于发送消息 ($OSQPost$) 和接收消息 ($OSQPend$) 所需的参数可从本节前述的代码中看出来, 因此下面对创建消息队列 ($OSQCreate$) 及其所需参数进行说明, $OSQCreate$ 的原型为: $OS_EVENT *OSQCreate(void **start, INT8U *flgTbl, INT16U size)$, 其中 $start$ 和 $size$ 与 μ C/OS-II 原来的参数的意义相同, 而 $flgTbl$ 将传递给 OS_Q 数据结构中的元素 $*OSQFlagPtr$, 用来记录 $MsgTbl$ 中对应位的信息是否已经被使用。

3 实验分析

对实验中应用逻辑做如下安排: task0 每隔 5s 向消息队列申请一个消息, 而 task1 每隔 3s 向消息队列发送一个消息。首先采用 $\mu\text{C}/\text{OS}-\text{II}$ 原有消息队列的处理方式以验证 1.2 中提出的 $\mu\text{C}/\text{OS}-\text{II}$ 中存在的数

据安全性问题, 其运行结果如图 3 所示。

```
task1 has send message:message0
task0 has received message:message0
task1 has send message:message1
task1 has send message:message2
task0 has received message:message2
task1 has send message:message3
task0 has received message:message3
task1 has send message:message4
task1 has send message:message5
task0 has received message:message5
```

图 3 使用 $\mu\text{C}/\text{OS}-\text{II}$ 原消息队列机制的运行结果

从图 3 可见由于 task0 没能在 task1 发送下一个消息之前从消息队列中取走 task1 前一次发送的消息, message2 和 message5 分别覆盖了 message1 和 message4。接下来采用改进后的消息队列处理方式来验证所做改进的正确性和有效性, 其运行结果如图 4 所示。

```
task1 has send message:message0
task0 has received message:message0
task1 has send message:message1
it's not safe now,I have to wait
task0 has received message:message1
task1 has send message:message2
task0 has received message:message2
task1 has send message:message3
it's not safe now,I have to wait
task0 has received message:message3
task1 has send message:message4
it's not safe now,I have to wait
task0 has received message:message4
task1 has send message:message5
task0 has received message:message5
```

图 4 使用改进后的消息队列机制的运行结果

从图 4 可见改进后的消息队列机制确实能保证数据的安全性, 这一安全性的保证需要应用函数在不确定修改消息是否安全时通过调用 OSQTest 函数进行

查询, 当然查询会额外消耗一定的时间, 但是从第 2 节中 OSQTest 的源代码可以看出这一查询时间是相当短的, 因此在安全性要求较高的环境中使用这一查询是值得的。

4 结束语

文中分析了 $\mu\text{C}/\text{OS}-\text{II}$ 中消息队列通信机制的实现原理以及其中存在的数据安全性问题, 通过改进消息队列通信涉及的数据结构, 增加相应的系统函数, 从而增强了 $\mu\text{C}/\text{OS}-\text{II}$ 中消息队列通信的数据安全性。最后通过实验证明以上改进在实际应用中的有效性。

参考文献:

- [1] 胡修林, 杨刚, 张蕴玉. 嵌入式多任务操作系统中的任务间通信策略[J]. 自动化技术与应用, 2004(7): 39-42.
- [2] 员青, 钱锋, 田蔚凤. 占先式实时内核 $\mu\text{C}/\text{OS}-\text{II}$ 在车辆动态监控/调度实验平台中的应用[J]. 电子测量技术, 2007(10): 146-149.
- [3] Labrosse J. 嵌入式实时操作系统 $\mu\text{C}/\text{OS}-\text{II}$ [M]. 邵贝贝等, 译. 北京: 北京航空航天大学出版社, 2003: 245-269.
- [4] 吴国民. $\mu\text{C}/\text{OS}-\text{II}$ 实现实时消息传递[J]. 现代计算机: 专业版, 2007(10): 132-134.
- [5] 周世杰, 刘锦德, 秦志光. 消息队列技术研究: 综述与一个实例[J]. 计算机科学, 2002(2): 84-86.
- [6] 朱方娥, 曹宝香. 基于 JMS 的消息队列中间件的研究与实现[J]. 计算机技术与发展, 2008, 18(5): 172-175.
- [7] Tai Stefan. Conditional Messaging: Extending Reliable Messaging with Application Conditions [C]//Proceedings of the 22nd International Conference on Distributed Computing Systems. [s. l.]: [s. n.], 2002: 123-132.
- [8] Chiang Mei-Ling, Li Yun-Chen. LyrNET: A zero-copy TCP/IP protocol stack for embedded operating systems [C]//11th IEEE International Embedded and Real-Time Computing Systems and Applications. [s. l.]: [s. n.], 2005: 123-128.
- [9] 李之堂, 李家春. 模糊神经网络在入侵检测中的应用[J]. 小型微型计算机系统, 2002, 23(10): 1235-1238.
- [10] Stolfo S J, Fan Wei, Lee Wenke, et al. Task description of Kddcup'99 [EB/OL]. 1999. <http://kdd.ics.uci.edu/databases/kddcup99/task.html>.

(上接第 150 页)

- on Security and Privacy. Oakland, CA: [s. n.], 1999.
- [4] Bass T. Intrusion detection systems and multisensor data fusion [J]. Communications of the ACM, 2000, 43(4): 99-105.
- [5] Giacinto G, Roli F, Didaci L. Fusion of multiple classifiers for intrusion detection in computer networks [J]. Pattern Recognition Letters, 2003, 24(12): 1795-1803.
- [6] 郭文普, 孙继银. 一种基于数据融合的分布式入侵检测系统[J]. 计算机技术与发展, 2006, 16(2): 217-219.
- [7] Lippmann R P, Cunningham R K. Improving intrusion detection performance using keyword selection and neural networks