软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(6):1631−1646 [doi: 10.13328/j.cnki.jos.006241]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向 SPARC 处理器架构的操作系统异常管理验证*

马 智[1], 乔 磊[1,3], 杨孟飞[2], 李少峰[1,4]

[1](北京控制工程研究所,北京 100190)

[2](中国空间技术研究院,北京 100094)

[3](计算机科学国家重点实验室(中国科学院 软件研究所),北京 100190)

[4](西安电子科技大学 计算机科学与技术学院,陕西 西安 710071)

通讯作者: 乔磊, E-mail: fly2moon@aliyun.com

摘 要: 航天器等安全关键系统是典型的嵌入式系统,具有多任务并发、中断频发等特点.操作系统是其最基础的软件,构建一个正确的操作系统是保障航天器系统高可信运行的关键.异常管理作为操作系统最底层的功能,负责引导系统控制流的突变来响应处理器状态中的某些变化,异常管理的正确性是整个操作系统正确性的基础.提出一种基于 Hoare-logic 的验证框架,用于证明面向 SPARC 处理器架构操作系统异常管理的正确性,特别针对多任务并发和中断频发实时操作系统异常嵌套与异常中发生任务切换的情况,将异常管理划分为 5 个阶段进行全面的形式化建模,并且在 Coq 证明定理辅助工具中实现了此框架.基于该框架,验证了我国北斗三号在轨实际应用的航天器嵌入式实时操作系统 SpaceOS 异常管理功能的正确性.

关键词: 操作系统;异常管理;异常嵌套;任务切换;形式化验证

中图法分类号: TP311

中文引用格式: 马智,乔磊,杨孟飞,李少峰.面向 SPARC 处理器架构的操作系统异常管理验证.软件学报,2021,32(6):1631−1646. http://www.jos.org.cn/1000-9825/6241.htm

英文引用格式: Ma Z, Qiao L, Yang MF, Li SF. Verification of operating system exception management for SPARC processor architecture. Ruan Jian Xue Bao/Journal of Software, 2021,32(6):1631−1646 (in Chinese). http://www.jos.org.cn/1000-9825/6241.htm

Verification of Operating System Exception Management for SPARC Processor Architecture

MA Zhi[1], QIAO Lei[1,3], YANG Meng-Fei[2], LI Shao-Feng[1,4]

[1](Beijing Institute of Control Engineering, Beijing 100190, China)

[2](China Academy of Space Technology, Beijing 100094, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[4](School of Computer Science and Technology, Xidian University, Xi'an 710071, China)

Abstract: Safety-critical systems such as spacecraft are typical embedded systems with the characteristics of multi-task concurrency and frequent interruptions. The operating system is the most fundamental software of computer, and building a correct operating system is crucial to ensure the reliability of the spacecraft system. Exception management, as the lowest level function of the operating system, is responsible for guiding sudden changes of control flow in response to certain changes in the processor state. Exception management is the basis for the correctness of the entire operating system correctness. This study proposes a verification framework based on Hoare-logic to

* 基金项目: 国家自然科学基金(61632005, 61802017, 62032004); 中国科学院软件研究所计算机科学国家重点实验室开放课题(SYSKF1804)

Foundation item: National Natural Science Foundation of China (61632005, 61802017, 62032004); Open project of State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences (SYSKF1804)

本文由"形式化方法与应用"专题特约编辑田聪教授推荐.

收稿时间: 2020-08-17; 修改时间: 2020-10-26; 采用时间: 2020-12-19; jos 在线出版时间: 2021-02-07

prove the correctness of exception management for SPARC processor architecture operating systems. Especially for multi-task concurrency and frequent interruption of real-time operating system exception nesting and task switching in exceptions, the exception management is divided into five stages for comprehensive formal modeling, and this framework is implemented in the Coq proving theorem assistant tool. Based on this framework, the correctness of the exception management function of the spacecraft embedded real-time operating system SpaceOS, which is actually used by China's Beidou-3, is verified.

Key words: operating system; exception management; exception nesting; task switching; formal verification

如今,计算机系统已经被广泛应用于航天、国防、医疗、金融等安全攸关领域,这类系统一旦发生错误,将会造成严重后果.建立高可信的计算机系统,在安全攸关领域是极其重要的.而操作系统作为计算机系统中最基础的底层软件,是构建高可信度计算机系统的关键.异常管理作为操作系统最底层的功能,负责引导系统控制流的突变来响应处理器状态中的某些变化.异常管理的正确性,是整个操作系统正确性的基础.传统的软件测试方法只能发现程序的执行结果是否正确,而不能证明程序本身有无错误.传统的测试方法无法保证程序正确性的百分百覆盖,在一些极端苛刻的环境之下,程序可能出现错误.为了解决这个问题,形式化验证方法逐渐成为研究的热点.形式化验证方法是一种确保软硬件系统可靠性和安全性的有效方法,它是基于严格数学基础的、对计算机软件系统进行形式规约、开发和验证的技术.其中,形式规约使用形式语言构建所开发软件系统的规约,它们对应于软件生命周期不同阶段的制品,刻画系统不同抽象层次的模型和性质,例如需求模型、设计模型甚至代码和代码的执行模型等.通过形式验证,证明不同形式规约之间的逻辑关系,这些逻辑关系反映了在软件开发不同阶段软件制品之间需要满足的各类正确性需求.例如,形式验证给出"系统设计模型满足若干特定性质"的证明构造[1].

1 相关工作

至今为止,国内外的相关研究已经对操作系统内核开展了大量的形式化验证工作,包括 NICTA 团队对 seL4 的验证[3]、Gerwin Klein 等人利用 Isabelle/HOL 对 seL4 内核进行模型层和代码层的形式化验证.内核模块的验证主要包含内存管理、并发管理、IO 管理等,同时证明了模型层与代码层的一致性.耶鲁大学的 Shao 团队提出了一个可扩展的体系结构 CertiKOS,用于构建经过形式化验证的并发 OS 内核[3,4],并发允许跨越不同的抽象层交错执行内核或用户模块,每一层都具有一组不同的可观察事件.作者采用分层组合的方法,以适合的抽象级别对每一层的可观察事件进行了形式化建模与验证.Feng 团队对 uC/OS-II 的验证[5,6]主要针对整个复杂软件系统整体的可靠性展开验证,考虑到 OS 内核不同功能模块的实现逻辑各不相同,作者针对各个模块的特点分别使用特定的验证方法来证明模块的可靠性,最后将这些模块连接在一起,验证整个系统的可靠性.BICE 的 Qiao 团队对 SpaceOS[7,8]进行了验证:在文献[7]中,Qiao 等人使用 Rodin 建模工具和 Event-B 数理抽象方法对 SpaceOS 内存管理的正确性进行了验证;文献[8]中,Jiang 等人使用 Coq 对 SpaceOS 操作系统任务管理功能的需求层提出了 4 条安全性质,并对其进行了建模与验证.

一个经典的实时嵌入式操作系统功能架构[9]如图 1 所示.

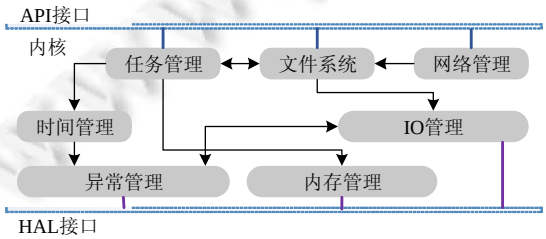

Fig.1 Classic real-time embedded operating system architecture

图 1 经典实时嵌入式操作系统架构

异常管理是操作系统最底层的功能,它的正确性决定了整个操作系统是否能够安全可靠地运行,因而被给

予了极大的重视.文献[10]中,作者提出了基于投影时序逻辑(PTL)的定义,并将这种定义推广到包含任意多中断事件的中断系统上,同时使用投影时序逻辑定义的基本中断语句扩充建模仿真和验证语言(MSVL),并扩展了 MSVL 语言的解释器,使其可以对嵌套中断系统进行建模仿真和验证.在文献[3]中,作者针对外部设备为 CertiKOS 带来的异步异常(中断)开展了形式化验证工作,构建了“虚拟”设备对象进行驱动程序验证,并使用了一种抽象中断模型,试图解决代码验证中由于中断而导致的不确定性.文献[11]中,作者利用多线程系统验证的方法来验证中断系统.由于中断缺乏形式化语义支持,所以使用该方法时,将中断的语义用等价的线程语义进行形式化描述.然而,中断和线程之间精细的差别,使得用线程语义描述中断这个过程变得棘手.文献[12]利用时间 Petri 网建模嵌套中断系统,并将 Petri 网所建立的系统模型转换为时间自动机,从而借助时间自动机理论进行模型检测.文献[13]中,针对于中断驱动系统中容易产生的时序错误,作者使用时间自动机对中断事件以及中断处理过程进行建模,并使用模型检测技术来进行可达性分析.

但遗憾的是,目前大部分关于异常的形式化研究,都是基于外部设备产生的异步异常(中断)带给操作系统的不确定性问题和时序问题为焦点来开展的,并没有对操作系统异常管理本身的正确性进行验证.本文重点关注实时操作系统异常管理功能的逻辑正确性,主要面临以下难点.

- 异常根据触发条件可分为同步与异步异常,面对两种不同类型的异常,异常管理功能的执行逻辑也不尽相同,两种不同的系统状态都需要进行考虑;
- 实时操作系统的内核可抢占特性,导致系统不仅仅要面对最基本的异常响应情况,还需要考虑异常嵌套、异常处理子程序(ESR)中发生任务切换[14]的情况;
- 面对基本异常的响应,异常管理在进行上下文保护和恢复的过程中,上下文内容始终保持一致.面对含有任务切换异常的响应,进行上下文恢复操作时的上下文内容已经发生了变化,在形式化建模过程中,需要对其进行全面覆盖.

针对以上难点,本文提出一种基于 Hoare-logic 的验证框架,用于证明面向 SPARC 处理器架构操作系统异常管理的正确性.第 2 节介绍操作系统的异常控制流和实时操作系统可能发生的 3 种异常的情况.第 3 节详细介绍异常管理验证框架的形式化建模工作,并且给出解决以上 3 个难点的详细方案.第 4 节借助半自动化证明工具 Coq,使用该验证框架对实际应用在北斗三号航天器系统上的 SpaceOS 操作系统异常管理的正确性进行验证.最后一节总结全文,并对未来研究提出进一步的展望.

2 基础知识

本节对基础知识进行介绍,其中,第 2.1 节对异常执行流以及实时操作系统面临的 3 种异常情况进行介绍,第 2.2 节对半自动化证明工具 Coq 进行介绍.

2.1 实时操作系统异常处理

现代计算机系统通过使控制流[15]发生突变来对某些突发情况做出反应,一般而言,我们把这些突变称为异常控制流(exceptional control flow,简称 ECF).异常控制流发生在计算机系统的各个层次,比如:在硬件层,硬件检测到的事件会触发控制突然转移到异常处理程序;在操作系统层,内核通过上下文切换控制一个用户任务转移到另一个用户任务;在应用层,一个任务可以发送信号到另一个任务,而接收者会将 CPU 控制权突然转移到它的一个信号处理程序.异常是异常控制流的一种形式,对它的管理一部分由硬件实现,一部分由操作系统实现.由操作系统实现的这一部分即为异常管理,由于和硬件密切联系,对于操作系统的异常管理设计将随硬件平台的不同而有所不同.异常根据触发事件的不同,一般被划分为两类:同步异常与异步异常.前者是执行一条指令的直接产物,通常被称为陷阱;后者则是由处理器外部 IO 设备产生,也被称为中断.

系统中可能发生的每一种类型的异常都被分配了唯一的非负整数种类的异常标号,其中一些标号是由处理器的设计者分配的,例如被零除、内存地址未对齐、窗口上溢等;其余的标号则是由操作系统的开发者进行分配,例如系统调用、任务切换和响应外部 IO 信号等.计算机启动时(重启或加电),操作系统会分配和初始化一张称为异常向量的跳转表[16],这张表中包含异常标号和对应的跳转地址.当系统在执行某个程序时,若处理器检

测到了即将要发生的异常事件,则会在响应异常时首先进入异常向量表寻找异常标号,随后转入异常管理进行指令跳转处理.

如图 2 所示,这是一个典型的异常处理流程:操作系统将 CPU 的运行抽象为任务的形式呈现给用户,任务 *i* 正常执行时接收到异常信号,然后对异常事件进行响应,对于异步异常的响应,会等待当前正在执行的指令完成之后才会进行响应,如图 2 中异步异常信号所示.系统响应异常信号后,控制流转入异常向量表之中寻找与之匹配的异常标号并进入正确的异常管理入口.异常管理需要完成的任务主要有 3 个:(1) 对任务上下文进行保护;(2) 引导系统控制流转入异常处理子程序(ESR),并设定正确的返回入口;(3) ESR 处理完毕后返回异常管理对任务上下文进行恢复.

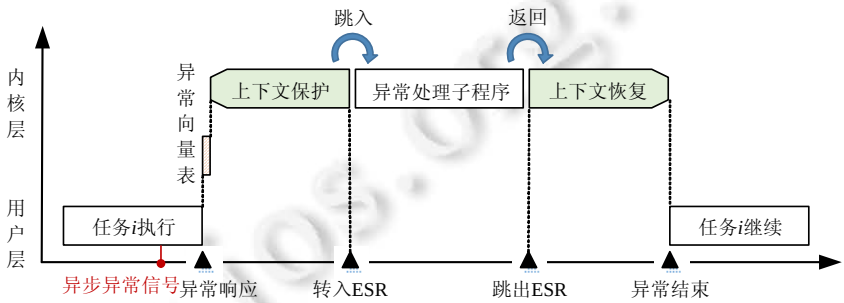

Fig.2 Basic exception

图 2 基本异常情况

为了保证对外部事件以及高优先级任务进行实时响应,实时 OS 对异常嵌套的情况也必须进行考虑.一个典型的异常嵌套如图 3 所示:在处理一个异常的过程中,通常会出现更紧迫的异常事件请求响应,因而导致了异常嵌套.值得注意的是:刚进入异常管理后,系统会自动屏蔽所有异常事件的响应.这样做的目的是为了防止系统正在保存上下文信息的时候被打断而产生数据丢失.同样的处理也被应用在上下文恢复中:当异常子程序处理完毕后,返回异常管理进行上下文恢复前,会屏蔽所有的异常响应.因此,允许更紧迫的异常事件被响应的情况只能发生在 ESR 正常执行中,如图 3 中所示,ESR1 在执行的过程中被高优先级 ESR2 打断.

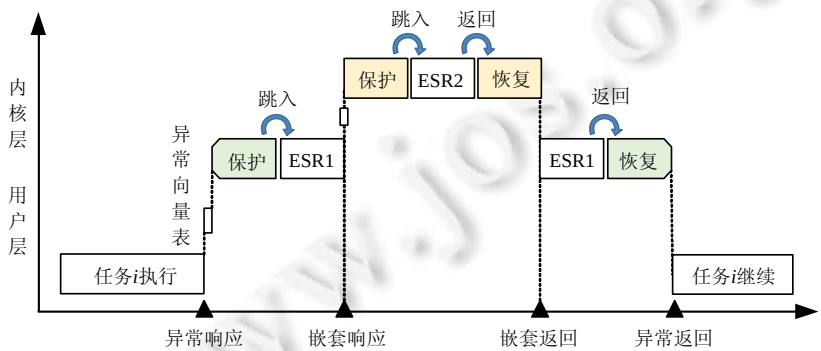

Fig.3 Nesting exception

图 3 异常嵌套情况

为了保障系统的实时性,实时 OS 的调度策略采用基于固定优先级的时间片轮转方法,也就意味着 CPU 是可抢占的.在异常处理子程序执行的过程中,往往会出现任务创建任务、停止任务操作,这导致系统会在异常管理流程中发生任务的切换.若在 ESR 中创建的任务优先级比当前任务高,则会在上下文恢复时被设置为就绪状态,当异常返回时,系统将根据情况进行一次任务调度去运行优先级最高的就绪任务,而不一定要继续运行被打断的原任务.如图 4 所示,这是一个高优先级抢占低优先级的情景:第 1 次异常响应,任务 *i* 被打断,控制流转入异

常管理对任务 *i* 的上下文进行保护,随即执行 ESR1;ESR1 执行过程中,将更高优先级的任务 *y* 设置为就绪态,执行完毕后返回异常管理;异常管理接收到任务切换的信号,将任务 *y* 的上下文信息拉取到寄存器中再退出异常管理,此时 CPU 被任务 *y* 抢占,所以系统会执行任务 *y* 而不是任务 *i*;当任务 *y* 执行完毕之后,调度器触发任务切换异常,进入异常管理中再次进行任务调度,ESR2 结束后返回异常管理进行任务切换,此时,恢复的上下文才是任务 *i* 中被保存的信息;最后退出异常管理,任务 *i* 继续执行.

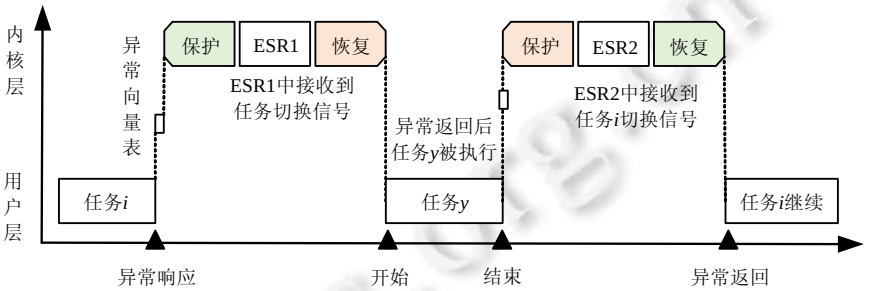

Fig.4 Task switching of exception

图 4 异常中发生任务切换

2.2 Coq形式化证明工具

Coq 是法国国家科学研究中心等机构研发的交互式定理证明辅助工具[17],形式化研究人员可以使用它来表示规范说明,开发满足规范说明的程序.Coq 非常适合于开发那些在金融、医疗、航空、航天等领域需要绝对可信的程序,这些领域中的程序需要严格地符合规范说明,需要对这些程序进行形式化验证.Coq 系统提供了一种具有很强表达能力的逻辑,通常被称为高阶逻辑.Coq 系统也可以被用作一个逻辑框架,在该框架下,我们可以为新的逻辑提供公理,并基于这些逻辑来开发证明.

Coq 的规范说明语言 Gallina 可以描述程序设计语言中的常用类型和程序.一个标识符的类型通常由声明和一些规则来描述,后者使用类型规则从较简单的类型出发构造符合类型,每个类型规则表达了类型的子部分同类型结构之间的联系.比如,在 Coq 中使用“Inductive”来归纳定义一种类型,归纳定义需要一个名称、一个类型、零个或多个规则来定义一个归纳类,每个规则都具有单独的名称,称之为“构造器”.如下文代码 1 所示,这是对于自然数的定义,其中:*O* 是一个自然数;*S* 是一个构造器,它需要一个自然数作为输入,输出则为自然数.其规则为:若 *n* 是一个自然数,那么 *S n* 同样为自然数.

代码 1. 在 Coq 中定义自然数

```
Inductive nat:Type:=
  |O
  |S(n:nat).
```

在 Coq 中,计算是通过对验证对象进行连续规约,直到得到一个不能归约的形式来完成的.因此,Coq 中计算总是可终止的(强正规化).可计算性方面的经典结论表明:一个可以描述所有可计算函数的程序设计语言的一个基本性质就是,必须对包含不停机函数的描述.所以,一些可计算函数能够在 Coq 中表示,但是并不可以通过归约机制执行.尽管有这些限制,Coq 的类型系统还是十分强大的,它是描述可计算函数的一个很大的子类.因此,要求强正规化并不会显著地减弱它的表达能力.计算机辅助定理证明工具是一个大家族,除了 Coq 之外,还有很多享有盛誉的工具,如 Automath,Nqthm,Mizar,LCF,Isabelle 等.我们之所以选择 Coq,是因为它具有一个与众不同的特性——可以从证明中生成可靠的程序和模块.

3 框架建模

第 2 节介绍了系统异常处理的 3 种需求,本节开展实时操作系统异常管理验证框架的建模工作.在计算机系统中,CPU 提供了运行动力,这种推动力最直接地体现在指令寄存器的改变上.当计算机上电后,复位逻辑电

路将一个固定的地址写入指令寄存器中,CPU 从该地址处读取并执行指令.在指令的执行过程中,CPU 硬件逻辑根据所取到的指令完成状态的改变,这种改变由两部分组成:(1) CPU 自身状态的改变,包括指令寄存器、通用寄存器、状态寄存器以及控制寄存等的改变;(2) 内存内容的改变,包括全局变量、任务堆栈等.指令寄存器的更新也就是通常所说的“指令地址加 1”或“转移地址生成”,系统运行的推动力正是由这种自动更新所形成的.在这种力的推动下,CPU 按顺序在指令序列上移动,我们称之为“执行流”.在执行流的流动过程中,CPU 不断地改变自身和外部的状态,这就是我们通常所说的“程序的运行”.因此,计算机的运行的状态可以使用一个两元组来表示,即⟨寄存器状态,内存状态⟩.为了对异常管理进行细粒度的建模,我们将两元组扩展为五元组:⟨*G*,*L*,*S*,*M*,*H*⟩,其中,*G*,*L*,*S* 表示不同类型的寄存器状态,*M*,*H* 表示不同类型的内存成员,在第 3.1 节中对其进行详细解释.

3.1 基本数据类型定义

寄存器的种类与使用方法与硬件平台密切相关,本文重点关注采用 SPARC 体系的操作系统,将严格按照 SPARC 标准对寄存器模型进行数学建模.SPARC 处理器中包含 32 个通用寄存器,其中有 8 个寄存器是全局寄存器,另外 24 个寄存器是窗口寄存器.这 24 个窗口寄存器可以被分为 3 组:out,in,local,SPARC 处理器中含有 8 个窗口,执行时将按照 0 到 7 的顺序依次切换窗口.在任务运行时,窗口寄存器中保存着当前任务的上下文信息.在任意时刻,任务只使用一组窗口寄存器.当发生系统异常时,处理器会在不同的寄存器窗口之间移动,以保存当前上下文环境.对全局寄存器和窗口寄存器的形式化定义如下所示:

$$
\begin{aligned}
&(GlobReg) \quad G ::= g0 \mid g1 \mid ... \mid g7\\
&(WindReg) \quad W ::= l0 \mid ... \mid l7 \mid i0 \mid ... \mid i7 \mid o0 \mid ... \mid o7\\
&(SpecReg) \quad S ::= psr \mid wim \mid tbr \mid pc \mid npc\\
&(PsrReg) \quad psr ::= PIL \mid CWP \mid ET\\
&(TbrReg) \quad tbr ::= TBA \mid tta \mid ttb\\
&(CWinReg) \quad L \in W \to S(CWP)
\end{aligned}
$$

其中,

- *G* 表示全局寄存器(*GlobReg*);
- *W* 表示窗口寄存器(*WindReg*);
- *S* 用来表示 SPARC 的特殊寄存器(*SpecReg*);
- *psr* 表示程序状态寄存器(*PsrReg*),*PsrReg*(processor state register)是一个 32 位的寄存器,它包含了 11 个字段,这些字段保存着处理器的状态信息.与异常管理密切相关的主要有 3 个字段:
  - *PSR_enable_traps*(*ET*)字段,表示处理器此时是否可以响应异常:当 *ET* 为 0 时,处理器处于异常屏蔽状态;*ET* 为 1 时,处理器处于异常响应状态;
  - *PSR_proc_interrupt_level*(*PIL*)表示处理器目前可以接受异常的最低优先级;
  - *PSR_current_window_pointer*(*CWP*)字段可以理解为指向当前寄存器窗口的指针,表示系统目前正在使用的寄存器窗口是哪一个;
- *wim* 为窗口无效屏蔽寄存器,表示窗口寄存器的使用状态,通常用来判断窗口溢出或下溢;
- *tbr* 为异常基址标志寄存器(*TbrReg*),用来表示异常的基本属性,其中,*TBA* 表示异常的标志号,*tta* 表示异常的种类(同步或异步),*ttb* 表示异常的优先级;
- *pc* 和 *npc* 表示程序计数器:*pc* 保存着系统正在执行的指令的地址,*npc* 保存着将要被执行的下一条指令的地址.

为了描述窗口寄存器中的旋转动作,我们使用 *L* 来表示当前窗口寄存器,CWP 指向的寄存器窗口即为系统当前使用的 *WindReg*,当 CWP 的值发生改变时,便伴随着窗口的旋转.图 5 中展示了窗口寄存器的这些特征,存在 8 个寄存器窗口,分别是 0 到 7(用 *w*0~*w*7 标记),每个窗口对应了 24 个寄存器,其中 16 个与相邻的窗口共用.这些寄存器窗口是交迭的,以循环圈的方式连接起来.SPARC 执行 RESTORE 和 SAVE 指令时,将导致窗口切换,CWP 指针用来指向当前正在使用的窗口.特权模式下的 RETT 指令(return from trap)和异常事件(TRAP 指令)

也将导致窗口的切换.图中右上角的 WIM 寄存器用来表示每个窗口被使用的状态,每个窗口在 WIM 中对应 1 个 bit 位.当寄存器窗口都被使用时,如果再发生一个额外的 SAVE 指令则会超过窗口的容量,此时将触发一个窗口寄存器溢出陷阱;与之相反,当发生 RESTORE 或 RETT 指令时,如果 WIM 寄存器对应 bit 位无效,此时将触发一个窗口寄存器下溢陷阱.

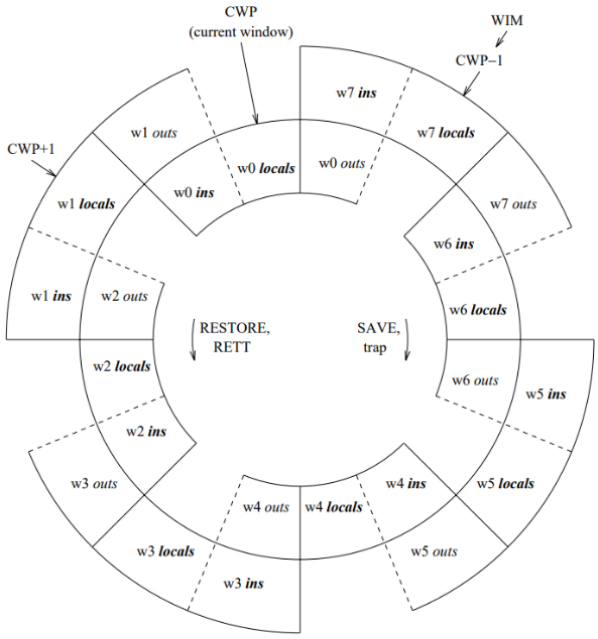

Fig.5 SPARC window register

图 5 SPARC 窗口寄存器

异常管理涉及的内存内容的改变由两部分组成:一是内存中的全局变量,二是内存堆栈.我们使用 M 来表示内存中的全局变量(*MemoVar*):

(*MemoVar*) $M ::= nest \mid tcb \mid ossm \mid swflag$

(*TcbAddr*) $tcb ::= CurTcb \mid PreTcb$

(*Offset*) $O ::= 0 \mid 4 \mid 8 \mid ... \mid 128$

(*HeapStack*) $H \in O \rightarrow sp$, **where** $sp = L(o6)$

其中,

- *nest* 表示异常嵌套次数;
- *tcb* 表示任务控制块指针,由 *CurTcb* 与 *PreTcb* 组成:前者指向当前运行任务的 *Tcb* 地址,后者指向前一个任务运行的 *Tcb* 地址.当任务发生切换时,会使用它们;
- *ossm* 表示异常信号表,异常信号表中保存着系统中所有异常处理函数的入口地址;
- *swflag* 用来判断 ESR 中是否发生了任务切换:若其值为 1,则发生任务切换;否则不发生;
- H 表示异常管理中使用到的内存堆栈地址;
- *sp* 表示栈顶指针,在 SPARC 中,使用当前窗口寄存器的 *o*6 表示 *sp* 的值;
- O 则表示异常管理中系统设置的地址偏移量.

计算机的运行过程中,系统的某些状态根据寄存器中内容的变换而改变,需要对这些状态进行形式化描述.程序状态寄存器中的 *ET* 位为 0 时,系统进入异常屏蔽状态;否则,系统正常响应异常.除了状态的定义,我们对异常管理中使用的一些基本操作在此进行定义,如打开异常响应(*en_Exce*)、关闭异常响应(*dis_Exce*)、寄存器窗口左移动 *win_TL*(*L*))、右移 *win_TR*(*L*)等:

$$S(ET=0)\Rightarrow exce\_disable \quad S(ET\neq 0)\Rightarrow exce\_enable$$
$$dis\_Exce(S)\overset{def}{=}S(ET)=0 \quad en\_Exce(S)\overset{def}{=}S(ET)=1$$
$$win\_TL(L)\overset{def}{=}(S(CWP)+1,W)\ \textbf{where}\ L=(S(CWP),W)$$
$$win\_TR(L)\overset{def}{=}(S(CWP)-1,W)\ \textbf{where}\ L=(S(CWP),W)$$

3.2 异常管理的状态迁移

本节开展异常管理系统的建模工作.为了对系统进行细粒度的建模,将系统的状态由⟨寄存器内容,内存内容⟩扩展为一个五元组:⟨*G*,*L*,*S*,*M*,*H*⟩,*G* 表示全局寄存器,*L* 表示当前窗口寄存器,*S* 表示特殊寄存器,*M* 代表内存全局变量,*H* 代表内存堆栈地址.将异常管理程序逻辑形式化框架建模分为 5 个阶段,分别是异常向量表处理阶段、上下文保护阶段、ESR 跳转阶段、上下文恢复阶段和异常退出阶段,5 个阶段在时序上是顺序执行的.在每一阶段中,我们使用结构化的操作语义对其进行建模并勾画出了对应的系统状态迁移图进行对比.

1) 异常向量表处理

异常发生之后,系统会根据 *ET* 位信息来决定是否对异常进行响应.

- 若系统处于异常屏蔽态,则忽略异常继续维持原本的状态;
- 若系统处于异常响应状态,则会根据不同的异常类型分别进行处理:对于同步异常(陷阱),系统立即进行响应;对于异步异常(中断),系统执行完当前指令之后才会响应.这种区别也导致中断返回时,需要将程序计数器指向当前指令的下一条.

系统响应异常后,控制流转入系统内核,首先进入异常向量表中进行基本数据处理,随后正式进入异常管理,这样做的目的是为所有异常提供一个统一的接口.异常向量表中的操作可以归纳为 3 步:首先关闭异常响应(*dis_Exce*),然后将 *psr* 中的数据存入 *l*0 之中(*save_psr*),最后使用 *save_pc_npc* 将程序计数器 *pc* 与 *npc* 存储到 *l*1 与 *l*2 中.*l*1 即是异常处理完毕后的指令返回地址,*l*2 存储的是返回时的下一条指令地址:

$$save\_pc\_npc(S,L)\overset{def}{=}S(pc)\rightsquigarrow L(l1),S(npc)\rightsquigarrow L(l2)$$
$$save\_psr(S,L)\overset{def}{=}S(psr)\rightsquigarrow L(l0)$$

异常向量表中系统的状态迁移定义为公式(1)与公式(2),主要涉及寄存器内容的改变,对应的状态迁移图如图 6 所示.公式(1)中异常响应处于关闭状态,系统不会对异常进行响应;公式(2)中异常响应处于开启状态,控制流转入异常向量表中执行操作(*evt*),最终跳转到异常管理入口(*exce*),此时系统处于异常屏蔽状态,这是为了防止接下来的上下文保护过程中出现数据丢失的情况:

$$\frac{exce\_disable \quad win\_TR(L)\rightarrow abort}{evt(G,L,S)\Rightarrow abort} \tag{1}$$

$$\frac{exce\_enable \quad dis\_Exce(S)\rightarrow S' \quad save\_pc\_npc(S',L')\rightarrow(S'',L'') \quad save\_psr(S'',L'')\rightarrow(S''',L''')}{evt(G,L,S)\Rightarrow exce(G,L''',S''') \quad exce\_disable} \tag{2}$$

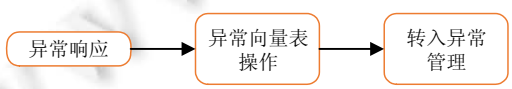

Fig.6 Exception vector table processing migration diagram

图 6 异常向量表处理迁移图

2) 上下文保护

进入异常管理首先需要检查是否发生窗口上溢,将检查窗口上溢函数定义为 *win_Ovf*,特殊寄存器 WIM 指向已经被系统使用过的窗口,如果接下来要使用的窗口(*CWP*-1)等于 WIM,系统进行窗口上溢处理,将异常管理中对窗口上溢的处理函数定义为 *save_WF*.具体的操作是把需要保存的窗口寄存器 *i*0~*i*7 和 *l*0~*l*7 中的数据保存

到对应的内存堆栈之中,保存完毕后,将 WIM 的数值右移一位,窗口寄存器的内容已经被保存,此时就可以对当前窗口寄存器进行数值改变等操作:

$$win\_Ovf \overset{\text{def}}{=} \begin{cases} \text{true}, & \text{if } S(CWP)-1=S(WIM) \\ \text{false}, & \text{if } S(CWP)-1\neq S(WIM) \end{cases}$$

$$save\_WF(L,S,H) \overset{\text{def}}{=} L(l0\sim l7,i0\sim i7) \rightsquigarrow H(sp,(0\sim 28,32\sim 60)),$$
$$\textbf{let } S(WIM) \rightsquigarrow l4 \textbf{ in } l4-1 \rightsquigarrow S(WIM)$$

如果没有发生窗口上溢,直接开始保存上下文环境,上下文内容即当前寄存器窗口的内容,需要保存的寄存器有 *l*0~*l*2,*i*0~*i*7 和全局寄存器 *g*1~*g*7.这是因为进入当前寄存器窗口之后,*l*3~*l*7,*o*0~*o*7 都还未使用过,*g*0 的值恒为 0,所以不需要保存,将保存上下文函数定义为 *save_Reg*,此时用于保存异常上下文的堆栈使用的是发生异常的任务的堆栈空间.为了保障系统的实时性,异常管理需要考虑异常嵌套的情况,使用全局变量 *nest* 来表示嵌套层数,初始值为 0,系统每响应一次异常,*nest* 数值加一:

$$save\_Reg(G,L,H) \overset{\text{def}}{=} \textbf{let } (sp = fp+128) \textbf{ in } [L\{l0\sim l2,i0\sim i7\},G\{g1\sim g7\}] \rightsquigarrow H\{sp,(0\sim 8,32\sim 60,64\sim 88)\}$$

$$plus\_Nest(M) \overset{\text{def}}{=} \textbf{let } M(nest) \rightsquigarrow l5 \textbf{ in } l5+1 \rightsquigarrow M(nest)$$

公式(3)和公式(4)为异常管理从初始状态到保护环境上下文状态的迁移过程,对应的迁移过程如图 7 所示:

$$\frac{win\_Ovf=\text{true}\ \ win\_TR(L)\to L'\ \ save\_WF(L',S,H)\to(L',S',H')\ \ win\_TL(L'')\to L'''\ \ save\_Reg(G,L''',H')\to(G,L''',H'')\ \ plus\_Nest(M)\to M'}{exce(G,L,S,M,H)\Rightarrow(G,L''',S',M',H'')\ \ exce\_disable} \tag{3}$$

$$\frac{win\_Ovf=\text{false}\ \ save\_Reg(G,L,H)\to(G,L',H')\ \ plus\_Nest(M)\to M'}{exce(G,L,M,H)\Rightarrow(G,L',M',H')\ \ exce\_disable} \tag{4}$$

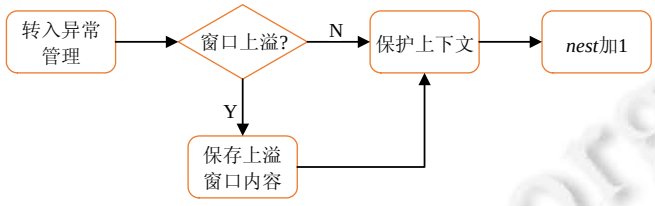

Fig.7 State transition diagram in the context protection phase

图 7 上下文保护阶段的状态迁移图

公式(3)中,系统若发生窗口上溢,将寄存器窗口向右移动一位,保存此窗口的寄存器内容到相应的堆栈之中,调整 WIM 寄存器的指向.处理完毕后,将寄存器窗口向左移动归位,开始保存当前窗口寄存器中的上下文环境,将寄存器中的内容保存到任务堆栈中.随后,异常嵌套加 1.值得注意的是:此时 ET 位信息并没有发生改变,系统仍然处于异常屏蔽状态.公式(4)中,系统没有发生窗口上溢,跳过窗口上溢处理步骤,直接开始保存当前窗口寄存器的上下文环境,嵌套层数 *nest* 加 1.

3) ESR 跳转

上下文保存工作完成之后,下一步便是引导控制流跳转入正确的 ESR 中.此时需要为任务切换做一些准备工作,这样做是因为 ESR 中可以执行任务启动、任务停止等操作来进行任务切换.SpaceOS 通过改变当前任务控制块指针(*CurTcb*)来切换任务,此时,*CurTcb* 指向被异常打断的任务.若发生任务切换,*CurTcb* 就会指向另外的任务,所以异常管理中还需要对控制块指针进行备份为任务切换做准备,将 *CurTcb* 中的内容存入 *PreTcb* 之中.

为了保护上下文的数据安全,进入异常管理后,系统会一直处于异常屏蔽状态,上下文保存完毕之后需要打开异常响应,调整 ET 位的数值,并且需要考虑异常嵌套的情况,即:在 ESR 运行过程中,只有高优先级的异常才可以打断低优先级的异常,异常管理通过设置 PIL 的值来反映这种状态.*set_Exce* 函数表示根据 *tbr* 寄存器中存储的异常基本信息调整系统状态,当异常类型为陷阱时,则屏蔽所有中断响应,此时 ESR 中只能对陷阱类型的异常

进行响应;当异常类型为中断时,设置 *pil* 位为它的优先级 *ttb*,只有高优先级才可以打断低优先级.这些操作大多由计算机硬件自动完成,异常管理之中只需要设置相应的寄存器状态即可:

$$store\_Tcb(L,M)\overset{\text{def}}{=}\textbf{let}\ M(CurTcb)\rightsquigarrow l4\ \textbf{in}\ l4\rightsquigarrow M(PreTcb)$$

$$set\_Exce_{pil}(S)=\begin{cases}S(pil)\leftarrow 0, & \text{if } tbr(tta)=trap\\ S(pil)\leftarrow tbr(ttb), & \text{if } tbr(tta)=inter\end{cases}$$

$$exce\_Nest\overset{\text{def}}{=}\begin{cases}\text{true}, & \text{if } M(nest=1)\\ \text{false}, & \text{if } M(nest\neq 1)\end{cases}$$

我们将进入 ESR 之前的准备状态定义为公式(5)和公式(6),系统的状态迁移图如图 8 所示.

$$\left.\frac{\begin{array}{c}exce\_disable\ \ exce\_Nest=\text{true}\\ store\_Tcb(L,M)\rightarrow(L',M')\ \ set\_Exce_{pil}(S)\rightarrow S'\ \ en\_Exce(S')\rightarrow S''\end{array}}{exce(L,M,S)\Rightarrow(L'',M'',S'')\ \ exce\_enable}\right\}\tag{5}$$

$$\frac{exce\_disable\ \ exce\_Nest\rightarrow\text{false}\ \ set\_Exce_{pil}(S)\rightarrow S'\ \ en\_Exce(S')\rightarrow S''}{exce(S)\Rightarrow(S'')\ \ exce\_enable}\tag{6}$$

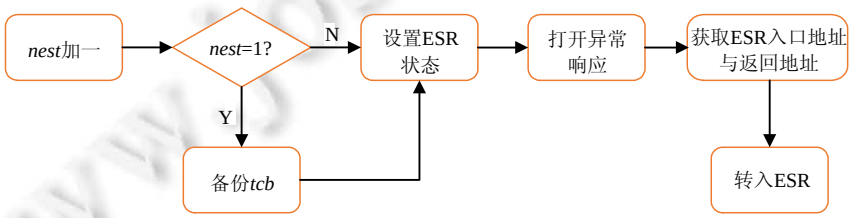

Fig.8 ESR jump stage migration diagram

图 8 ESR 跳转阶段迁移图

公式(5)中,如果异常嵌套层数为一(*exce_Nest*),备份当前任务控制块指针(*store_Tcb*),根据异常基本属性设置系统状态,最后打开异常响应;公式(6)表示若异常嵌套层数不为一,即此时系统中发生了多个异常并且正在处理的异常不是第 1 个发生的异常,这时候不需要对 *Tcb* 信息进行保存,仅仅根据异常的属性来调整系统对其他异常的响应条件即可.

系统状态调整完毕后,异常管理会引导控制流跳转到 ESR 中,寻找 ESR 入口地址操作定义为 *find_Exceaddr*,*Tbr* 寄存器中的 TBA 保存着异常的标志号,*ossm* 表示异常标志表,保存了每个异常标志所对应的 ESR 入口地址,因此可以根据异常的标志信息在标志表中寻找对应的 ESR 地址,将此地址保存在 *l*6 中.ESR 处理完成后,控制流跳转回异常管理,需要一个返回地址,将跳转前 *pc* 指向的地址保存到 *o*7 之中,定义为函数 *pass_PC*.这一阶段的状态迁移定义为公式(7):

$$\left.\frac{\begin{array}{l}find\_Exceaddr(L,S,M)\overset{\text{def}}{=}\textbf{let}\ L(l3)\leftarrow S(TBA)\ \textbf{in}\ M(ossm(l3))\rightarrow L(l6)\\ pass\_PC(L,S)\overset{\text{def}}{=}S(pc)\rightarrow L(o7)\\ findExceaddr(L,S,M)\rightarrow(L',S',M)\ \ passPC(L',S')\rightarrow(L'',S'')\end{array}}{exce(L,M,S)\Rightarrow esr(L'',M,S'')\ \ exce\_enable}\right\}\tag{7}$$

4) 上下文恢复

ESR 执行完毕后,控制流跳转回异常管理,在进行上下文恢复前,需要考虑异常嵌套和任务切换的情况.

- 首先判断异常是否嵌套,只有在异常非嵌套的情况下才会发生任务的切换;
- 然后判断 ESR 中是否发生任务切换:若不发生,直接进行上下文恢复操作;若发生任务切换,需要保存原任务使用过的所有寄存器窗口,否则任务一旦切换这些内容都会丢失.要注意的是,此时上下文恢复的

内容也转变为了新任务的上下文.

这一阶段重点关注发生任务切换时,异常管理的处理情况.系统根据全局变量 *swflag* 来判断 ESR 中是否发生了任务切换:若其值为 1,则发生任务切换;否则结果相反,将其状态表示定义为 *SW*.我们将寄存器窗口保存定义为 *sw_aWReg*,保存被打断的任务所使用过的所有寄存器窗口.这里以一个具体的例子进行说明:假设 CWP 此时指向窗口 4,WIM 指向窗口 7,则说明任务使用过的窗口为 4~6.保存上下文过程中,*save_Reg* 已经将窗口 4 的内容存放到了任务堆栈之中,所以此时 *sw_aWReg* 只需要保存窗口 5 和窗口 6.具体的操作如以下公式所示:

$$SW \overset{\text{def}}{=} \begin{cases} \text{true}, & \text{if } swflag = 1 \\ \text{false}, & \text{if } swflag = 0 \end{cases}$$

$$switch\_Ts(M,L) \overset{\text{def}}{=} CurTcb \rightsquigarrow sp$$

$$Wim\_Cwp(G,S) \overset{\text{def}}{=} \begin{cases} \text{true} \\ \text{false} \end{cases} \textbf{ let } CWP \rightsquigarrow g4, WIM \rightsquigarrow g7 \textbf{ in } \begin{cases} \text{if } g4+1 \neq g7 \\ \text{if } g4+1 = g7 \end{cases}$$

$$rest\_Reg(G,L,H) \overset{\text{def}}{=} H\{sp,(0 \sim 8, 32 \sim 60, 64 \sim 88)\} \rightsquigarrow [L\{l0 \sim l2, i0 \sim i7\}, G\{g1 \sim g7\}]$$

$$sw\_aWReg(G,L,S,H) \overset{\text{def}}{=} \begin{cases} \text{let } winTL(L) = (S(CWP)+1, W) \text{ in} \\ \text{let } sp+128 \text{ in} \\ L(o0 \sim o7, l0 \sim l7, i0 \sim i7) \rightsquigarrow \\ H\{sp,(0 \sim 28, 32 \sim 60, 64 \sim 92)\}, & \text{if } Wim\_Cwp = \text{true} \\ \perp, & \text{if } Wim\_Cwp = \text{false} \end{cases}$$

当 *Wim_Cwp* 的状态为 true 时,将窗口左移即 CWP 加 1,随后保存当前窗口的所有内容到对应的内存堆栈之中,全局寄存器的内容通过 *save_Reg* 已经被保存,所以可以通过全局寄存器来充当数据交换的中介;当 *Wim_Cwp* 状态为 false 时,说明使用过的寄存器窗口已经全部保存完毕.要注意的是,窗口 4 的恢复机制与窗口 5、窗口 6 不同:窗口 4 会在异常管理恢复上下文的过程中进行处理;窗口 5 与窗口 6 则在此任务继续执行以后,发生窗口上溢时在上溢处理中恢复内容.保存原任务使用过的寄存器窗口之后,接下来需要考虑将新任务的上下文恢复入寄存器中.在 ESR 处理过程中,系统的 *Tcb* 指针已经指向了新任务地址,也就说,*CurTcb* 已经发生了改变,这也是为什么我们要在上下文保护阶段中备份 *CurTcb* 的原因.如果要恢复新任务的上下文,则需要将 *CurTcb* 中的地址信息存储到 *sp* 寄存器中,然后再进行恢复操作,将其定义为 *switch_Ts*.恢复寄存器上下文为保护上下文 *save_Reg* 的逆函数,定义为 *rest_Reg*.

我们使用公式(8)~公式(10)来表示图 9 中的状态迁移过程:

$$\frac{exce\_enable \;\; dis\_Exce(S) \rightarrow S' \;\; exce\_Nest = \text{false} \;\; rest\_Reg(G,L,H) \rightarrow (G',L',H)}{exce(G,L,S,M,H) \Rightarrow (G',L',S',M,H) \;\; exce\_disable} \tag{8}$$

$$\frac{exce\_enable \;\; dis\_Exce(S) \rightarrow S' \;\; exce\_Nest = \text{true} \;\; SW = \text{false} \;\; rest\_Reg(G,L,H) \rightarrow (G',L',H)}{exce(G,L,S,M,H) \Rightarrow (G',L',S',M,H) \;\; exce\_disable} \tag{9}$$

$$\left.\frac{\begin{array}{c} exce\_enable \;\; dis\_Exce(S) \rightarrow S' \;\; exce\_Nest = \text{true} \;\; SW = \text{true} \\ sw\_aWReg(G,L,S',H) \rightarrow (G',L',S'',H') \;\; switch\_Ts(M,L') \rightarrow (M,L'') \;\; restReg(G',L'',H') \rightarrow (G'',L''',H') \end{array}}{exce(G,L,S,M,H) \Rightarrow (G'',L''',S'',M,H') \;\; exce\_disable}\right\} \tag{10}$$

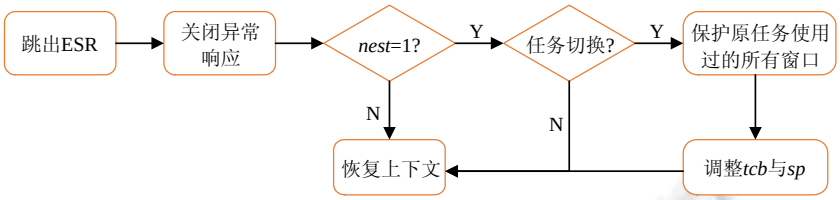

Fig.9 State transition diagram of context recovery phase

图 9 上下文恢复阶段状态迁移图

跳出 ESR 后,控制流返回异常管理中.为了保护数据安全关闭异常响应,进行异常嵌套判断.

- 若嵌套层数不为 1,则直接恢复当前窗口上下文,此时恢复的上下文为被打断的 ESR 中的上下文,使用公式(8)来表示这一过程;
- 若嵌套层数为 1,表明此时系统中所有的 ESR 都已经执行完毕,再次进行判断,是否发生任务切换:如果不发生任务切换,则进行上下文恢复,此时恢复的上下文便是原任务的上下文,使用公式(9)来表示这一过程;若在 ESR 之中发生了任务切换,需要执行 *swtich_aWreg* 操作,然后将新任务的上下文地址存入 *sp* 之中,最后恢复新任务的上下文到处理器中,这一过程使用公式(10)来表示.

对于 ESR 中不发生任务切换的情况来说,此时系统中的 CWP 与 WIM 与进入异常处理时的初始状态是一样的,所以 *sp* 中的存储的始终是保存上下文的堆栈地址;而对于发生任务切换的情况,*sp* 的值是从任务控制块中获取的,所以进行上下文恢复时恢复的是新任务的数据.

5) 异常退出

至此,异常管理的工作迎来尾声,控制流即将从异常管理跳出.退出之前需要检查异常退出时是否会发生窗口下溢:如果会,则将发生下溢的寄存器窗口内容从堆栈之中恢复出来,并且改变 WIM 的值.判断异常退出时是否会发生窗口下溢被定义为 *win_Odf*,将 CWP 的值加 1 与 WIM 相比较:如果相等,则说明会发生窗口下溢;如果不等,则说明不会发生窗口下溢:

$$win\_Odf \overset{\text{def}}{=} \begin{cases} \text{true}, & \text{if } S(CWP)+1=S(WIM) \\ \text{false}, & \text{if } S(CWP)+1\neq S(WIM) \end{cases}.$$

对窗口下溢的处理定义为 *rest_WF*,从对应的堆栈地址中恢复出原有的寄存器窗口数据,可以看作它是 *save_WF* 的逆函数.将异常嵌套层数减少定义为 *redu_Nest*,ESR 执行前嵌套层数加一来表示系统中目前有多个异常在执行.同样,当系统中的 ESR 执行完毕之后,再需要将异常嵌套层数减一.异常返回时,程序状态也需要和原来保持一致,*save_psr* 将 *psr* 寄存器中的内容存储到了 *l*0 之中,在异常退出前需要将其还原,被定义为 *rest_psr*:

$$rest\_WF(L,S,H) \overset{\text{def}}{=} \begin{cases} \text{let } winTL(L)=(S(CWP)+1,W) \text{ in} \\ \text{let } S(WIM) \rightsquigarrow o4 \text{ in } o4+1 \rightsquigarrow S(WIM) \\ H(sp,(0\sim 28,32\sim 60)) \rightsquigarrow L(l0\sim l7,i0\sim i7) \\ winTR(L)=(S(CWP)-1,W) \end{cases}$$

$$redu\_Nest(M) \overset{\text{def}}{=} \textbf{let } M(nest) \rightsquigarrow o5 \textbf{ in } o5+1 \rightsquigarrow M(nest)$$

$$rest\_psr(L,S) \overset{\text{def}}{=} L(l0) \rightsquigarrow S(psr)$$

我们使用公式(11)、公式(12)来表示这一阶段的迁移过程,状态迁移图如图 10 所示.

$$\left.\frac{exce\_disable \;\; win\_Odf=\text{true} \;\; rest\_WF(L,S,H)\rightarrow(L',S',H)}{redu\_Nest(L',M)\rightarrow(L'',M'') \;\; rest\_psr(L'',S')\rightarrow(L'',S'') \;\; en\_Exce(S'')\rightarrow S'''}\right\} \tag{11}$$
$$exce(L,S,M,H)\Rightarrow(L'',S''',M'',H) \;\; en\_disable$$

$$\left.\frac{exce\_disable \;\; win\_Odf=\text{false}}{redu\_Nest(L,M)\rightarrow(L',M') \;\; rest\_psr(L',S)\rightarrow(L',S') \;\; en\_Exce(S')\rightarrow S''}\right\} \tag{12}$$
$$exce(L,S,M)\Rightarrow(L',S'',M') \;\; en\_disable$$

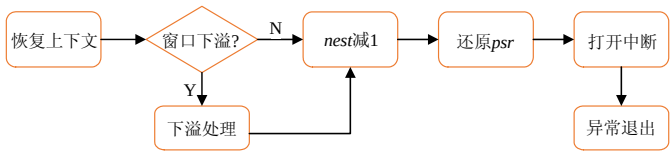

Fig.10 State transition diagram of abnormal exit phase

图 10 异常退出阶段状态迁移图

其中,

- 公式(11)中,发生窗口下溢,异常管理进行下溢处理,左移寄存器窗口,还原堆栈中内容;然后右移寄存器窗口,调整 WIM 的指向,嵌套层数减一,改变内容中的全局变量 *nest*,还原 *psr*,保持退出时程序状态寄存器与进入异常时的状态一致;最后开打异常,使系统可以正常响应异常,最后异常退出;
- 公式(12)中,窗口下溢不发生,则直接跳过下溢处理,执行后续的操作.

4 实例验证

本节使用前文描述的形式化验证框架对北斗三号[18]在轨实际应用的航天器实时操作系统 SpaceOS 异常管理功能的正确性进行证明,SpaceOS 是我国第 1 个自主研发并进行空间飞行的航天器嵌入式实时操作系统,系统资源占用低、实时性强,具有多任务并发、中断频发等特点,主要功能包括内存管理、异常管理、任务管理、IO 管理等.如图 11 是北斗三号任务实际运行情况,一个控制周期中包含 6 个控制任务、两个异常嵌套、6 次异常中任务切换.任务 1 运行时被异常 *A* 打断,高优先级异常 *B* 打断 *A* 的处理过程并启动任务 2,异常 *A* 在上下文恢复阶段将任务 2 设为就绪态;异常退出时,处理器执行任务 2;任务 2 执行完毕后再次进行任务切换,执行任务 3.以此类推,当任务 6 执行完毕后,通过异常 *G* 将处理器分配给任务 1 继续执行.

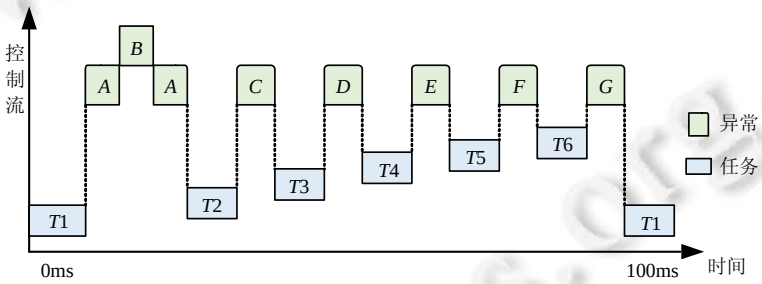

Fig.11 Beidou 3 mission operation example

图 11 北斗三号任务运行实例

程序逻辑是描述和论证程序行为的逻辑,程序和逻辑有着本质的联系,如果把程序看成一个执行过程,程序逻辑的基本方法是先建立程序和逻辑间联系的形式化方法,然后建立程序逻辑系统,并在此系统中研究程序的各种性质.在此给出异常管理程序逻辑的数学规范——$P\{S\}Q$,其中,

- *S* 代表程序逻辑,由第 3 节中介绍的 12 条公式组成;
- *P* 和 *Q* 是有关程序变元的逻辑表达式,*P* 称为前置条件,表明执行程序逻辑之前程序变量应具有的性质,可看作程序正确执行的前提;*Q* 称为后置条件,刻画了语句结束执行时的系统状态,可以看作 *S* 应实现的逻辑结果.

然后对异常管理的前置条件和后置条件进行分析与刻画,根据异常管理中是否会产生任务切换分为两种情况.

- 当 ESR 中含有任务的启动或停止操作时会产生任务的切换,异常退出时,寄存器中的内容是新任务的上下文数据,由 *switch_Ts* 函数实现;并且需要将原任务使用过的所有寄存器窗口都保存到堆栈之中,由 *rest_WF* 函数实现;

- 当 ESR 中不包含任务的启动或停止操作时不会产生任务的切换,异常退出时,寄存器中的内容依旧是原任务的上下文数据.

将系统的初始状态定义为 *mac_Status_pre*,即异常响应即将进入中断向量表进行处理时刻的状态,任务初始状态也是 Hoare-logic 中的前置条件 *P*:

$$mac\_Status\_pre \overset{def}{=} G_s\{g1 \sim g7\} \wedge L_s\{CWP,(l0 \sim l7,i0 \sim i7,o0 \sim o7)\} \wedge S_s\{psr,tbr,wim,pc,npc\} \wedge M_s\{nest,tcb\},$$

其中,

- G_s 表示的全局寄存器的初始状态;
- L_s 表示进入异常管理时 *CWP* 指向的当前窗口状态;
- S_s 表示进入异常管理时刻特殊寄存器的状态,其中:*tbr* 表示记录了异常的类型与优先级,*wim* 表示原任务目前已经使用的寄存器窗口都有哪些,*pc* 保存的是被打断指令的地址;
- M_s 表示内存中全局变量在此时刻的值,其中:*nest* 表示系统中目前发生的异常嵌套的层数,为了方便讲述建模过程,我们假设目前系统中还并没有进行异常处理,即 *nest* 的值为 0;*tcb* 是任务控制块指针,此时指向的是原任务的代码入口地址.

异常退出时刻,系统的状态为异常管理程序逻辑框架的终止状态,即 Hoare-logic 中的后置条件,若异常中不发生任务切换,即 *swflag* 的值为 0,后置条件被定义为 *mac_Status_post*,也就是说,最终异常退出时,寄存器中的数据和响应异常时中的大部分数据始终保持一致,指令计数器的值 *pc*/*npc* 均指向下一条要被指向的指令,全局变量 *tcb* 依旧指向被异常打断的原任务.若 ESR 中有任务切换的情况,异常退出时刻的寄存器中存储的是新任务的上下文,后置条件被定义为 *mac_Status_new*,用 G_n,L_n 表示窗口寄存器中的新任务的上下文内容,通过函数 *sw_aWReg* 已经将原任务的使用过的所有窗口寄存器保存到了对应的堆栈之中,所以此时所有的窗口寄存器都可以被新任务使用.新任务的特殊寄存器的状态被定义为 S_n,内容中全局变量 *tcb* 中的指针不再指向原任务而是新任务:

$$mac\_Status\_post \overset{def}{=} G_s\{g1 \sim g7\} \wedge L_s\{CWP,(l0 \sim l7,i0 \sim i7,o0 \sim o7)\} \wedge S_s\{psr,tbr,wim,pc,npc\} \wedge M_s\{nest,tcb\}$$

$$mac\_Status\_new \overset{def}{=} G_n\{g1 \sim g7\} \wedge L_n\{CWP,(l0 \sim l7,i0 \sim i7,o0 \sim o7)\} \wedge S_n\{psr,tbr,wim,pc,npc\} \wedge M_n\{nest,tcb\}$$

定理 1(任务切换异常正确性). 如果有前置条件 *P*:*mac_Status_pre*(*exce*) *swflag*=1,则通过{*S*}中的有限步,可以得到后置条件 *Q*:*mac_Status_new*(*exce*).

下文代码 2 给出了定理 1 在 Coq 中的证明,前置条件即为异常响应时刻的初始状态,被定义为 *mac_Status_pre*;后置条件即为异常返回时刻的终止状态,被定义为 *mac_Status_new*.“*f0*”是异常管理程序逻辑执行的起始地址,而 *exce_design_code* 是异常管理在 coq 中的定义.可以通过我们所定义的基本数据类型和遗产管理逻辑推理公式来证明定理 1 成立.

代码 2. 定理 1 在 Coq 中的证明.

```
Theorem Switch_Exce_Proof:forall(ex:Exce)(sw:Flags),

    {mac_Status_pre (ex)∧(sw=1)}
    f0:exce_design_code
    {mac_Status_new(ex)}.

  Proof.
  …
  Qed.
```

定理 2(非切换异常正确性). 如果有前置条件 *P*:*mac_Status_pre*(*exce*) *swflag*=0,则通过{*S*}中的有限步,可以得到后置条件 *Q*:*mac_Status_post*(*exce*).

与定理 1 类似,我们使用了同样的方法对定理 2 进行了证明,如下文代码 3 所示,此处不再赘述.

我们借助于定理证明辅助工具 Coq 对 SpaceOS 异常管理的正确性进行形式化验证,表 1 中给出了我们验证 SpaceOS 异常管理的 Coq 代码行数统计.

代码 3. 定理 2 在 Coq 中的证明.

```
Theorem Normal_Exce_Proof:forall(ex:Exce)(sw:Flags),

    {mac_Status_pre(ex)∧(sw=0)}
    f0:exce_design_code
    {mac_Status_post(ex)}.

Proof.
…
Qed.
```

Table 1 Coq code line statistics

表 1 Coq 代码行数统计

功能描述	代码行数
数据类型定义	536
异常向量表	218
上下文保护	462
ESR 跳转	571
上下文恢复	1 183
异常退出	347
定理验证	2 674
总计	5 964
代码运行时间	58.24s

5 总 结

本文提出了一种基于 Hoare-logic 的验证框架,用于证明面向 SPARC 处理器架构的实时嵌入式操作系统异常管理功能的正确性.首先,对操作系统异常控制流进行了介绍,并且总结出了 3 种实时操作系统所面临的异常情况:基本异常、异常嵌套、伴随任务切换的异常;然后,使用结构化的操作语义搭建异常管理验证框架,将其划分为 5 个阶段:异常向量表处理、上下文保护、ESR 跳转、上下文恢复、异常退出,并对每个阶段的建模过程进行了详细的介绍;最后,使用此形式化框架对应用在北斗三号航天器系统上的 SpaceOS 操作系统进行验证,在 Coq 定理证明工具的帮助下,验证了 SpaceOS 异常管理的正确性.

目前,关于 SpaceOS 的形式化工作都是分模块展开的.在今后的工作中,我们将尝试把任务管理、内存管理和异常管理的正确性证明工作形式化的集成起来,形成系统整体安全性和正确性的证明.

References:

[1] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019,30(1):33−61 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[2] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engerlhardt K, Kolanski R, Norrish M, Sewell T. seL4: Formal verification of an OS kernel. In: Proc. of the ACM SIGOPS 22nd Symp. on Operating Systems Principles. 2009. 207−220.

[3] Chen H, Wu XN, Shao Z, Lockerman J, Gu RH. Toward compositional verification of interruptible OS kernels and device drivers. In: Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2016). New York: Association for Computing Machinery, 2016. 431−447.

[4] Gu RH, Shao Z, Chen H, Wu XN, Kim J, Sjöberg V, Costanzo D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation (OSDI 2016). USENIX Association, 2016. 653−669.

[5] Feng XY, Shao Z, Dong Y, Guo Y. Certifying low-level programs with hardware interrupts and preemptive threads. In: Proc. of the 29th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2008). New York: Association for Computing Machinery, 2008. 170−182.

[6] Feng XF, Shao Z, Guo Y, Dong Y. Combining domain-specific and foundational logics to verify complete software systems. In: Proc. of the VSTTE 2008. 2008. 54−69.

[7] Qiao L, Yang MF, Tan YL, Pu GG, Yang H. Formal verification of memory management system in spacecraft using Event-B. Ruan Jian Xue Bao/Journal of Software, 2017,28(5):1204−1220 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5218.htm [doi: 10.13328/j.cnki.jos.005218]

[8] Jiang JJ, Qiao L, Yang MF, Yang H, Liu B. Operating system task management requirements layer modeling and verification based on Coq. Ruan Jian Xue Bao/Journal of Software, 2020,31(8):2375−2387 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5961.htm [doi: 10.13328/j.cnki.jos.005961]

[9] Regehr J, Cooprider N. Interrupt verification via thread verification. Electronic Notes in Theoretical Computer Science, 2007, 174(9):139−150. [doi: 10.1016/j.entcs.2007.04.002]

[10] Cui J, Duan ZH, Tian C, Zhang N. Modeling and analysis of nested interrupt systems. Ruan Jian Xue Bao/Journal of Software, 2018,29(6):1670−1680 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5472.htm [doi: 10.13328/j.cnki.jos.005472]

[11] Liu H, Zhang H, Jiang Y, et al. iDola: Bridge modeling to verification and implementation of interrupt-driven systems. In: Proc. of the Theoretical Aspects of Software Engineering Conf. (TASE). IEEE, 2014. 193−200. [doi: 10.1109/TASE.2014.33]

[12] Alur R, Dill DL. A theory of timed automata. Theoretical Computer Science, 1994,126(2):183−235.

[13] Zhou XY, Gu B, Zhao JH, Yang MF, Li XD. Model checking technique for interrupt-driven system. Ruan Jian Xue Bao/Journal of Software, 2015,26(9):2212−2230 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4713.htm [doi: 10.13328/j.cnki.jos.004713]

[14] Najumudheen ESF, Mall R, Samanta D. Modeling and coverage analysis of programs with exception handling. In: Proc. of the 12th Innovations on Software Engineering Conf., Formerly Known as India Software Engineering Conf. (ISEC 2019). Article 15. New York: Association for Computing Machinery, 2019. 1−11. [doi: https://doi.org/10.1145/3299771.3299785]

[15] Robillard MP, Murphy GC. Static analysis to support the evolution of exception structure in object-oriented systems. ACM Trans. on Softw. Eng. Methodol, 2003,12(2):191−221. [doi: https://doi.org/10.1145/941566.941569]

[16] Harr R, Kaptelinin V. Interrupting or not: Exploring the effect of social context on interrupters' decision making. In: Proc. of the 7th Nordic Conf. on Human-Computer Interaction: Making Sense through Design (NordiCHI 2012). New York: Association for Computing Machinery, 2012. 707−710. [doi: https://doi.org/10.1145/2399016.2399124]

[17] Sozeau M, Boulier S, Forster Y, Tabareau N, Winterhalter T. Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. In: Proc. of the ACM Program. Lang. 4, POPL, Article 8. 2020. 28. [doi: https://doi.org/10.1145/3371076]

[18] Yang YX, Xu YY, Li JL, Yang C. Progress and performance evaluation of Bei Dou global navigation satellite system: Data analysis based on BDS-3 demonstration system. Scientia Sinica (Terrae), 2018,48(5):584−594 (in Chinese with English abstract).

附中文参考文献:

[1] 王戟,詹乃军,冯新宇,刘志明.形式化方法概貌.软件学报,2019,30(1):33−61. http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652]

[7] 乔磊,杨孟飞,谭彦亮,蒲戈光,杨桦.基于 Event-B 的航天器内存管理系统形式化验证.软件学报,2017,28(5):1204−1220. http://www.jos.org.cn/1000-9825/5218.htm [doi: 10.13328/j.cnki.jos.005218]

[8] 姜菁菁,乔磊,杨孟飞,杨桦,刘波.基于Coq的操作系统任务管理需求层建模及验证.软件学报,2020,31(8):2375−2387. http://www.jos.org.cn/1000-9825/5961.htm [doi: 10.13328/j.cnki.jos.005961]

[10] 崔进,段振华,田聪,张南.一种嵌套中断系统的建模和分析方法.软件学报,2018,29(6):1670−1680. http://www.jos.org.cn/1000-9825/5472.htm [doi: 10.13328/j.cnki.jos.005472]

[13] 周筱羽,顾斌,赵建华,杨孟飞,李宣东.中断驱动系统模型检验.软件学报,2015,26(9):2212−2230. http://www.jos.org.cn/1000-9825/4713.htm [doi: 10.13328/j.cnki.jos.004713]

[18] 杨元喜,许扬胤,李金龙,杨诚.北斗三号系统进展及性能预测——实验验证数据分析.中国科学(地球科学),2018,48(5):584−594.

马智(1994−),男,博士生,主要研究领域为空间飞行器嵌入式系统,控制系统,总体技术.

杨孟飞(1962−),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为空间飞行器嵌入式系统,控制系统,总体技术.

乔磊(1982−),男,博士,研究员,CCF 专业会员,主要研究领域为操作系统模型设计,存储系统,文件系统.

李少峰(1992−),男,博士生,主要研究领域为操作系统内存管理,文件系统.