|  |  |
| --- | --- |
| 时间戳 | 时间戳表示SVMK启动以来的时间值，每次启动时会对该值进行清0。 |
| tick | tick是系统的基本时间单位，是定时等待时间、时间窗口时间、的基本时间度量单位 |
| 核心态 | 在DeltaUnity中核心态是系统特权级状态，程序在此状态下可以执行特权级操作 |
| 用户态 | 在DeltaUnity中用户态是系统的非特权级状态，程序在此状态下只可以执行用户级操作 |
| 核心分区 | 运行在核心态的分区 |
| 应用分区 | 运行在用户态的分区 |
| IDLE分区 | 运行在核心态，是系统的空闲分区 |
| 用户空间 | 用户态程序可以访问的空间 |
| 核心空间 | 核心态程序可以访问的空间 |

/\*

分区：appVM，kerVM，idlVM

<NAME,ID>

ID

状态：休眠、就绪、运行、等待、挂起、挂等

分区管理：

\*/

Machine APPVMADT{

appVM;

kerVM;

idlVM;

appVMLogicAddress;//应用分区的逻辑地址范围

canAccessInterfaceSet;// 分区所配特权接口集合

exceptionVMSet;//产生异常的分区集合

schedulingTable;//时间调度表

SSTWakenUpType;//时间调度表唤醒类型，包含基本、定时、时间窗或时间调度表启动唤醒类型

A0=ω(vm'0, uwpVMID'0){1RK.2.1.T\_VMK\_ReturnCode vmkGetVMID(

T\_UBYTE \*ubpName,

T\_UWORD \*uwpVMID})}

A1=ω(vm'0, uwpVMID'0){

If(ubpName==NULL){

\*uwpVMID=curVM<ID>;

return VMK\_OK;}}{Q1}

A2= ubpName==NULL→ω(vm'0, uwpVMID'0){ uwpVMID=curVM<ID>;

return VMK\_OK;}{Q1}

A3=ω(vm'0, uwpVMID'0){ uwpVMID=curVM<ID>;return VMK\_OK;}

→ uwpVMID'1== curVM<ID>∧vmkGetVMID( )= VMK\_OK

A4= ubpName==NULL→ uwpVMID'1== curVM<ID>∧vmkGetVMID( )= VMK\_OK

A5= ubpName!=NULL→ω(vm'0, uwpVMID'0){Q1}

A6=ω(vm'0, uwpVMID'0){Q1}

→ω(vm'0){if(NoValidName(ubpName)){

return VMK\_INVALID\_ADDRESS;}}{Q2}

A7= ω(vm'0, uwpVMID'0){if(NoValidName(ubpName)){return VMK\_INVALID\_ADDRESS;}}{Q2}

→( (NoValidName(ubpName) →ω(vm'0, uwpVMID'0){ return VMK\_INVALID\_ADDRESS;})

A8= ubpName!=NULL

→( (NoValidName(ubpName) →ω(vm'0, uwpVMID'0){ return VMK\_INVALID\_ADDRESS;})

A9=ω(vm'0, uwpVMID'0){ return VMK\_INVALID\_ADDRESS;})

→ vmkGetVMID()=VMK\_INVALID\_ADDRESS

A10= ubpName!=NULL∧NoValidName(ubpName) → vmkGetVMID()=VMK\_INVALID\_ADDRESS

A11= ω(vm'0, uwpVMID'0){if(NoValidName(ubpName)){return VMK\_INVALID\_ADDRESS;}}{Q2}

→( !(NoValidName(ubpName) →ω(vm'0, uwpVMID'0){Q2})

A12= ubpName!=NULL∧ !(NoValidName(ubpName)

→ω(vm'0, uwpVMID'0){Q2})

A13= ubpName!=NULL∧ !(NoValidName(ubpName)

→ω(vm'0, uwpVMID'0){ if(vm.<NAME:ubpName>∈appVM){

return VMK\_INVALID\_NAME;}}){Q3}

A14=ω(vm'0, uwpVMID'0){ if(vm.<NAME:ubpName>∉appVM){

return VMK\_INVALID\_NAME;}}){Q3}

→ vm'1.<NAME:ubpName>∉appVM→ω(vm'1, uwpVMID'0){ return VMK\_INVALID\_NAME }

A15=ω(vm'1, uwpVMID'0){ return VMK\_INVALID\_NAME }

→ vmkGetVMID= VMK\_INVALID\_NAME

A16= ubpName!=NULL∧ !(NoValidName(ubpName) ∧ vm'1.<NAME:ubpName>∉appVM

→ vmkGetVMID= VMK\_INVALID\_NAME

A17=ω(vm'0, uwpVMID'0){ if(vm.<NAME:ubpName>∉appVM){

return VMK\_INVALID\_NAME;}}){Q3}

→ vm'1.<NAME:ubpName>∈appVM→ω(vm'1, uwpVMID'0){ Q3}

A18= ubpName!=NULL∧!(NoValidName(ubpName)

→ vm'1.<NAME:ubpName>∈appVM→ω(vm'1, uwpVMID'0){ Q3}

A19=ω(vm'1, uwpVMID'0){Q3}

→\*uwpVMID==vm.<ID>∧ω(vm'1, uwpVMID'1){return VMK\_OK;}

A20=ω(vm'1, uwpVMID'1){return VMK\_OK;}

→ vmkGetVMID( )= VMK\_OK

A21= ubpName!=NULL∧!(NoValidName(ubpName)∧ vm'1.<NAME:ubpName>∈appVM

→\* uwpVMID'1==vm.<ID>∧ vmkGetVMID( )= VMK\_OK

A4= ubpName==NULL→ \*uwpVMID'1== curVM<ID>∧vmkGetVMID( )= VMK\_OK

A10= ubpName!=NULL∧NoValidName(ubpName) → vmkGetVMID()=VMK\_INVALID\_ADDRESS

A16= ubpName!=NULL∧!(NoValidName(ubpName) ∧vm'1.<NAME:ubpName>∉appVM

→ vmkGetVMID= VMK\_INVALID\_NAME

//获取应用分区的ID号

1RK.2.1.T\_VMK\_ReturnCode vmkGetVMID(

T\_UBYTE \*ubpName,

T\_UWORD \*uwpVMID

){

If(ubpName==NULL){

\*uwpVMID=curVM<ID>;

return VMK\_OK;

}

if(NoValidName(ubpName)){

return VMK\_INVALID\_ADDRESS;

}

if(vm.<NAME:ubpName>∈appVM){

return VMK\_INVALID\_NAME;

}

\*uwpVMID=vm.<ID>;

return VMK\_OK;

}

//获取应用分区信息

RK.2.3 T\_VMK\_ReturnCode vmkGetVMInfo(

T\_UWORD uwVMID,

T\_VMK\_VMBasicInfo \*tpBuf

){

if(uwVMID ==VMK\_NULL\_VID){

\*tpBuf =curVM<INFO>;

return VMK\_OK;

}

vm.<ID: uwVMID >∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

\*tpBuf= vm.<INFO>;

if(NoValidvmInfo(tpBuf)){

return VMK\_INVALID\_ADDRESS;

}

return VMK\_OK;

}

//获取应用分区的名字

2RK.2.4.T\_VMK\_ReturnCode vmkGetVMName(

T\_UWORD uwVMID,

T\_UBYTE \*ubpName

){

If(uwVMID ==VMK\_NULL\_VID){

ubpName = curVM. < NAME >;

return VMK\_OK;

}

vm.<ID:uwVMID>∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

\*ubpName=copy(vm.<NAME>);

return VMK\_OK;

}

//获取逻辑地址对应的物理地址

RK.2.6 T\_VMK\_ReturnCode vmkGetVMPhysicalAddress(

T\_UWORD uwVMID,

T\_UWORD uwLogicalAddress,

T\_VOID\* vpPhyAdd;

){

vm.<ID:uwVMID>∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

If(uwLogicalAddress ∈appVMLogicAddress){

\* vpPhyAdd=getPhyAddress(uwVMID, uwLogicalAddress);

Return VMK\_OK

}else{

Return VMK\_FAIL;

}

}

//获取应用分区的状态

3RK.2.7.T\_VMK\_ReturnCode vmkGetVMStatus(

T\_UWORD uwVMID,

T\_VMK\_VMStatus \*tpStatus

){

If(uwVMID ==VMK\_NULL\_VID){

\*tpStatus= curVM.<STATUS>;

return VMK\_OK;

}

vm.<ID:uwVMID>∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

\* tpStatus = vm.< STATUS >;

return VMK\_OK;

}

//获取发生异常应用分区的ID号

RK.2.11 T\_VMK\_ReturnCode vmkGetExceptionVM(

T\_UWORD \*uwVMID;

){

If(curVM.accessInterface∉ canAccessInterfaceSet){

Return VMK\_INVALID\_USER;

}

vm∈exceptionVMSet;

if(vm==NULL){

return VMK\_INVALID\_STATE;

}

If(vm.isDebugging()){

return VMK\_INVALID\_USER;

}

If(curVM.isNotManagePartition(vm)){

Return VMK\_INVALID\_USER;

}

\*uwVMID= vm.<ID>;

Return VMK\_OK;

}

//获取时间调度表ID号

RK.2.13 T\_VMK\_ReturnCode vmkGetTTSID(

T\_UBYTE \*ubpTTSName,

T\_UWORD \*uwpTTSID;

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){

Return VMK\_INVALID\_USER;

}

If(ubpName==NULL){

If(curSchedulingTable==NULL){

Return VMK\_INVALID\_STATE;

}

\*uwpTTSID =curSchedulingTable.<ID>;

Return VMK\_OK;

}

If(noValidName(ubpTTSName)){

Return VMK\_INVALID\_ADDRESS;

}

tts.<NAME: ubpTTSName >∈ schedulingTable;

if(tts==NULL){

return VMK\_INVALID\_NAME;

}

\* uwpTTSID = tts.<ID>;

return VMK\_OK;

}

//获取时间调度表信息

RK.2.14 T\_VMK\_ReturnCode vmkGetTTSInfo(

T\_UWORD uwTTSID,

T\_VMK\_TTSBasicInfo \*tpBuf

){

If(uwTTSID == VMK\_NULL\_TTS){

If(curSchedulingTable==NULL){

Return VMK\_INVALID\_STATE;

}

\* tpBuf =curSchedulingTable.< Info >;

Return VMK\_OK;

}

tts.<ID: uwTTSID >∈ schedulingTable;

if(tts==NULL){

return VMK\_INVALID\_ ID;

}

\* tpBuf = tts.< Info>;

return VMK\_OK;

}

//获取时间调度表名字

RK.2.15 T\_VMK\_ReturnCode vmkGetTTSName(

T\_UWORD uwTTSID,

T\_UBYTE \*ubpTTSName

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){

Return VMK\_INVALID\_USER;

}

If(noValidName(ubpTTSName)){

Return VMK\_INVALID\_ADDRESS;

}

If(uwTTSID == VMK\_NULL\_TTS){

If(curSchedulingTable==NULL){

Return VMK\_INVALID\_STATE;

}

\*uwpTTSID =curSchedulingTable.<NAME>;

Return VMK\_OK;

}

tts.<NAME: uwTTSID >∈ schedulingTable;

if(tts==NULL){

return VMK\_INVALID\_ ID;

}

\* uwpTTSID = tts.< NAME>;

return VMK\_OK;

}

RK.2.19 T\_VMK\_ReturnCode vmkInGetVMCount(

T\_UWORD \*uwpVMCount

){

\*uwpVMCount=getSize(appVM);

If(\*uwpVMCount==NULL){

return VMK\_INVALID\_ADDRESS;

}

return VMK\_OK;

}

//使当前应用分区放弃CPU，根据指定的唤醒类型来唤醒该分区

interruptSet;// 未处理的虚拟外部中断或虚拟服务中断集合

waitingQueue; //差分时间等待队列集合

readyQueue; //就绪队列；

schedulingTable

RK.2.21 T\_VMK\_ReturnCode vmkHaltVM(

T\_VMK\_WakeupType tWakeupType,

T\_UWORD uwTicks

){

If(calledInSuspend()){//如果在中断处理程序中执行，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

If(curVM∈ idlVM){

Return VMK\_INVALID\_USER;

}

If(tWakeupType ==timeAwaken ){// 唤醒类型为定时唤醒类型时

If(uwTicks ==0){

Return VMK\_INVALID\_TIME;

}

clearCurVMStatus(READY);//清除当前分区的就绪状态

waitingQueue= waitingQueue∪<curVM, uwTicks >;//将当前分区插入到差分时间等待队列

CurVM.setCurVMStatus(WAITING);// 设置当前分区状态为等待状态

CurVM.setCurVMWakenUpType(tWakeupType);//设置当前分区的唤醒类型为<tWakeupType>

CurVM.callScheduling();

return VMK\_OK;

}

If(tWakeupType==SSTAwakenType ){// 唤醒类型为时间调度表调度启动唤醒类型

If( curVM. TTSInfo<TTSType>!=PriorityScheduleType){// 当前分区的调度类型不是优先级调度

Return VMK\_INVALID\_TYPE;

}

readQueue= readQueue-<curVM>;//将当前分区从优先级就绪队列中移出；

CurVM.setCurVMScheduling(TTSScheduleType);//设置当前分区调度类型为时间调度表调度；

CurVM.setCurVMStatus(READY);//设置当前分区状态为就绪状态；

CurVM.setCurVMWakenUpType(SSTAwakenType);//设置当前分区的唤醒类型为时间调度表启动唤醒类型；

CurVM.callScheduling();//请求分区调度；

return VMK\_OK;

}

If(tWakeupType== timeWindowAwakenType){// 唤醒类型为时间窗唤醒类型

If( curVM. TTSInfo<TTSType>!= TTSScheduleType){// 当前分区的调度类型不是时间调度表调度

Return VMK\_INVALID\_TYPE;

}

CurVM.giveupCPU();

CurVM.setSchedulingTable(curVM,idlVM); //并设置时间调度表当前分区为IDLE分区；

CurVM.recordWindowTime();//记录当前分区被停止的时间窗口

CurVM.setCurVMWakenUpType(timeWindowAwakenType);//设置当前分区的唤醒类型为时间窗唤醒

CurVM.callScheduling();//请求分区调度；

return VMK\_OK;

}

If(tWakeupType ==basicAwaken ){// 唤醒类型为基本唤醒类型时

If(interruptSet!=NULL){

Return VMK\_OK;

}

If( watingTime==0){

CurVM.clearCurVMStatus(READY);//清除当前分区的就绪状态

CurVM.setCurVMStatus(WAITING);// 设置当前分区状态为等待状态

CurVM.setCurVMWakenUpType(basicAwaken);//设置当前分区的唤醒类型为基本唤醒类型

CurVM.callScheduling();

return VMK\_OK;

}

CurVM.clearCurVMStatus(READY);//清除当前分区的就绪状态

waitingQueue= waitingQueue∪<curVM, uwTicks >;//将当前分区插入到差分时间等待队列

CurVM.setCurVMStatus(WAITING);// 设置当前分区状态为等待状态

CurVM.setCurVMWakenUpType(tWakeupType);//设置当前分区的唤醒类型为<tWakeupType>

CurVM.callScheduling();

Return Return VMK\_INVALID\_TIME;

}

}

//启动应用分区

schedulingType//调度集合，默认调度类型、优先级调度类型或时间调度表调度类型

RK.2.27 T\_VMK\_ReturnCode vmkStartVM(

T\_UWORD uwVMID,

T\_VMK\_SchedType tType

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(type∉schedulingType){// 如果<tType>不表示默认调度类型、优先级调度类型或时间调度表调度类型

Return VMK\_INVALID\_TYPE;

}

If(vm.isOnStarting()){//如果<uwVMID>指定的分区已经被启动

Return VMK\_INVALID\_STATE;

}

If(tType== VMK\_SCHED\_DEFAULT){// 调度类型为默认调度类型

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态

vm.callScheduling(DEFAULT)//采用用户所配置的调度类型或上一次运行的调度类型请求分区调度；

Return VMK\_OK;

}

If(tType== VMK\_SCHED\_PRIORITY || tType== VMK\_SCHED\_TIME\_TABLE){// 如果调度类型为优先级调度类型或时间调度表调度类型

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态

vm.callScheduling(tType);//根据调度类型<tType>请求分区调度

Return VMK\_OK;

}

}

//停止应用分区

RK.2.29 T\_VMK\_ReturnCode vmkStopVM(

T\_UWORD uwVMID

){

If(curVM.accessInterface ∉ curVM.canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

vm.<ID: uwVMID >∈ appVM;

if(vm!=NULL){// <uwVMId>指定的分区不是应用分区

return VMK\_INVALID\_ID;

}

if(vm.<STATUS:vmStatus>==unStarting){// 如果<uwVMId>指定的分区尚未启动

return VMK\_INVALID\_STATE；

}

if(vm.<STATUS:vmStatus>==READY){// 如果<uwVMId>指定的分区的状态为就绪态

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm .callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==WAITING){// 如果<uwVMId>指定的分区的状态是等待态

vm.setCurWaitingTime(0);//无论<uwVMId>指定的分区的等待时间是否超时，均使其等待时间为0；

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==HUNGUP){//如果<uwVMId>指定的分区处于挂起态

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.havaSecondException()){//如果<uwVMId>指定的分区产生了二次异常

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.callScheduling();//请求分区调度

return VMK\_OK;

}

}

//重启应用分区

RK.2.31 T\_VMK\_ReturnCode vmkResetVM(

T\_UWORD uwVMID,

T\_VMK\_SchedType tType

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

vm.<ID: uwVMID >∈ appVM;

if(vm!=NULL){// <uwVMId>指定的分区不是应用分区

return VMK\_INVALID\_ID;

}

If(type∉schedulingType){// 如果<tType>不表示默认调度类型、优先级调度类型或时间调度表调度类型

Return VMK\_INVALID\_TYPE;

}

if(vm.<STATUS:vmStatus>==unStarting){// 如果<uwVMId>指定的分区尚未启动

return VMK\_INVALID\_STATE；

}

If(uwVMID==VMK\_NULL\_VID){

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

vm.callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

If(vm.<STATUS:vmStatus>==READY){

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

vm.callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==WAITING){// 如果<uwVMId>指定的分区的状态是等待态

vm.setCurWaitingTime(0);//无论<uwVMId>指定的分区的等待时间是否超时，均使其等待时间为0；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

vm.callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==HUNGUP){//如果<uwVMId>指定的分区处于挂起态

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

vm.callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.havaSecondException()){//如果<uwVMId>指定的分区产生了二次异常

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

vm.callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

}

//挂起应用分区

RK.2.33 T\_VMK\_ReturnCode vmkSuspendVM(

T\_UWORD uwVMID

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

if(vm.<STATUS:vmStatus>==unStarting || vm.<STATUS:vmStatus>==HUNGUP){// 如果<uwVMId>指定的分区尚未启动或者如果应用分区已经挂起

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>==READY){// 如果<uwVMID>指定的分区为就绪态

vm.setCurStatus(HUNGUP);//将<uwVMID>指定分区的状态置为挂起态；

vm.callScheduling();//请求分区调度

return VMK\_OK;

}else{

vm.setCurStatusFlag(HUNGUP)；//设置应用分区挂起态标志位，如果应用分区已经是等待状态，则此时应用分区处于挂等态；

if(vm.curStatus==WATING){

vm.setCurStatusFlag(HUNGUPandWATING)；

}

return VMK\_OK;

}

}

//解挂应用分区

RK.2.35 T\_VMK\_ReturnCode vmkResumeVM(

T\_UWORD uwVMID

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(vm.isNotHUNGUP()){//应用分区尚未被挂起

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>== HUNGUPandWAITING){// 如果<uwVMID>指定的分区为挂等状态

vm.clearCurStatusFlag(HUNGUP)；

vm.setCurStatus(WAITING);// 清除该应用分区的挂起状态，使该应用分区处于等待状态；

return VMK\_OK;

}

If(vm.<STATUS:vmStatus>== HUNGUP){// 如果<uwVMID>指定的分区为挂起状态

vm.setCurStatus(READY);// 将其设置为就绪态；

callScheduling();//请求分区调度

return VMK\_OK;

}

}

//停止时间调度表

RK.2.38 T\_VMK\_ReturnCode vmkStopTTS(

T\_VOID

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

If(vm.schdulingTableIsNotOn()){//如果当前没有启动时间调度表调度

Return VMK\_INVALID\_ STATE;

}

//如果当前启动了时间调度表调度，那么依次执行以下操作：

Vm.setCurSchedulingTableStatus(wating);// 设置被停止的当前时间调度表的状态为等待主时间帧；

Vm.stopCurScheduling();

setCurSchedulingTableWindowStatus(IDLE);// 设置时间调度表调度的当前时间窗口分区为IDLE分区；

callScheduling();//请求分区调度

return VMK\_OK;

}

//启动时间调度表

RK.2.39 T\_VMK\_ReturnCode vmkStartTTS(

T\_UWORD uwTTSID

){

If(curVM.accessInterface ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

schedulingSST.<ID: uwpVMID >∈ schedulingTable //指定的时间调度表;

if(schedulingSST ==NULL){// 如果不存在<uwTTSID>指定的时间调度表，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

if(schedulingSST .isOnStarting()){//如果<uwTTSID>指定的时间调度表已经启动，应返回VMK\_INVALID\_STATE

return VMK\_INVALID\_ID STATE;

}else{

If(schedulingSST.timeManageStatus()==one-shot){

Vm.stopCurSchedulingTable();//停止当前时间调度表

schedulingSST.setCurSchedulingTableStatus(wating);// 设置<uwTTSID>指定的时间调度表的状态为等待主时间帧；

Vm.setCurSchedulingTableWindowStatus(IDLE);// 设置时间调度表调度的当前时间窗口分区为IDLE分区；

Vm.clearTimer();//清空定时器中断响应的间隔时间；

schedulingSST.setCurSchedulingTable(curSchedulingTable);// 设置<uwTTSID>指定的时间调度表为当前时间调度表；

Vm.setEnableTimerWindow();//使能定时器中断响应以当前分区时间窗口大小的间隔时间响应中断

Vm.timerOperation();//进行时间通知，包括系统tick时间、时间调度表窗口时间、时间调度表主帧时间

Vm.callScheduling();//请求分区调度

return VMK\_OK;

}else{

Vm.stopCurSchedulingTable();//停止当前时间调度表

schedulingSST.setCurSchedulingTableStatus(wating);// 设置<uwTTSID>指定的时间调度表的状态为等待主时间帧；

setCurSchedulingTableWindowStatus(IDLE);// 设置时间调度表调度的当前时间窗口分区为IDLE分区；

Vm.setCurTimerWindowType(watingTimer);// 设置当前时间窗口通知类型为等待主时间帧；

schedulingSST.setCurSchedulingTable(curSchedulingTable);// 设置<uwTTSID>指定的时间调度表为当前时间调度表；

Vm.timerOperation();/进行时间调度表主帧的时间通知；

Vm.callScheduling();//请求分区调度

return VMK\_OK;

}

}

}

RK.6.1 T\_VMK\_ReturnCode vmkInitializeVint(

T\_VMK\_VMVintEnMask \*tpVIntEnMask,

T\_VMK\_VINT\_HANDLER tpHandler

){

If(checkValidInterruptAdd(tpVIntEnMask)){// 指向的虚拟中断屏蔽码空间没有被映射、指向的虚拟中断屏蔽码空间不可读写访问

Return VMK\_INVALID\_ADDRESS;

}

If(checkValidInterruptAdd(tpHandler)){// 表示的地址没有被映射、表示的地址不可访问

Return VMK\_INVALID\_ADDRESS;

}

If(vm.tpInterruptIsInitialed()){//如果当前分区的虚拟中断已经初始化

Return VMK\_INVALID\_STATE;

}

If(checkNotValidAddress(tpHandler)){// 如果<tpHandler>表示的地址未按4字节对齐，应返回VMK\_INVALID\_ALIGNED

Return VMK\_INVALID\_ALIGNED;

}

Vm.instalInterruptEntryandMask(tpVIntEnMask, tpHandler);// 根据<tpVIntEnMask>和<tpHandler>安装当前分区的虚拟中断入口程序和屏蔽码

Vm.setTPInterruptStatus(TRUE);// 设置当前分区的虚拟中断初始化标识为TRUE

Return VMK\_OK;

}

PICset//虚拟中断号范围集合

PICHardwareset//实际的中断号范围集合

PICNotAllowed//当前分区屏蔽的外部中断集合

PICMapSet//中断映射表集合

RK.6.2 T\_VMK\_ReturnCode vmkDisablePIC(

T\_UWORD uwVIntNum

){

If(uwVIntNum∉PICset){// 如果<uwVIntNum>不在虚拟中断号范围内，应返回VMK\_INVALID\_INDEX

Return VMK\_INVALID\_INDEX;

}

Pic.<ID: uwVIntNum >∈PICNotAllowed;

If(pic!=NULL){// 如果虚拟中断号<uwVIntNum>对应的外部中断被当前分区屏蔽，应返回VMK\_MASKED

Return VMK\_MASKED;

}

picHardNum.<ID: uwVIntNum >∈PIDMapSet; 根据用户所配置的中断映射表将虚拟中断号<uwVIntNum>转换成硬件中断号

if(checkNoValid(picHardNum)){//如果无法将虚拟中断号映射到硬件中断号或者转换后的硬件中断号大于等于硬件中断个数

return VMK\_INVALID\_STATE;

}

If(picHardNum ∉ PICHardwareset){// 如果用户所配置中断映射表中硬件中断号不在实际的中断号范围内

Return VMK\_INVALID\_INDEX;

}

If(vm.disablePIDSuccessed()){//如果成功禁止当前分区的指定虚拟外部中断

Return VMK\_OK;

}

}

RK.6.3 T\_VMK\_ReturnCode vmkEnablePIC(

T\_UWORD uwVIntNum

){

If(uwVIntNum∉PICset){// 如果<uwVIntNum>不在虚拟中断号范围内，应返回VMK\_INVALID\_INDEX

Return VMK\_INVALID\_INDEX;

}

Pic.<ID: uwVIntNum >∈PIDNotAllowed;

If(pic!=NULL){// 如果虚拟中断号<uwVIntNum>对应的外部中断被当前分区屏蔽，应返回VMK\_MASKED

Return VMK\_MASKED;

}

picHardNum.<ID: uwVIntNum >∈PIDMapSet; 根据用户所配置的中断映射表将虚拟中断号<uwVIntNum>转换成硬件中断号

if(changeInValid()){//如果无法将虚拟中断号映射到硬件中断号或者转换后的硬件中断号大于等于硬件中断个数

return VMK\_INVALID\_STATE;

}

If(picHardNum ∉ PICHardwareset){// 如果用户所配置中断映射表中硬件中断号不在实际的中断号范围内

Return VMK\_INVALID\_INDEX;

}

If(enablePIDSuccessed()){//如果成功禁止当前分区的指定虚拟外部中断

Return VMK\_OK;

}

}

waitingTimeQueue/差分时间等待队列

RK.6.4 T\_VMK\_ReturnCode vmkSendServiceInt(

T\_UWORD uwVMID,

T\_UWORD uwServiceIntNum

){

If(vmkStopVM ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(uwVIntNum∉PICset){// 如果<uwVIntNum>不在虚拟中断号范围内，应返回VMK\_INVALID\_INDEX

Return VMK\_INVALID\_INDEX;

}

If(vm.isNotOnStarting()){//如果<uwVMID>指定的分区尚未被启动

Return VMK\_INVALID\_STATE

}

If(vm.tpInterruptIsNotInitialed()){//如果当前分区的虚拟中断已经初始化

Return VMK\_INVALID\_STATE;

}

If(vm.awakenType()!=VMK\_WAKEUP\_BASIC){// 定的分区的唤醒类型不是VMK\_WAKEUP\_BASIC

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

return VMK\_OK;

}else{

If(waitingTime==0){// 并且等待时间为0

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

awaken(vm);// 唤醒该分区

if(vm.isNotHungup()){//如果该分区没有被挂起，应设置该分区为就绪态

vm.setCurStatus(READY);

}

callScheduling();//请求分区调度

return VMK\_OK;

}

If(waitingTime<=ENDTIME){// 并且等待时间还未超时

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

waitingTimeQueue =waitingTimeQueue-vm;// 将该分区从差分时间等待队列上移除

awaken(vm);// 唤醒该分区

if(vm.isNotHungup()){//如果该分区没有被挂起，应设置该分区为就绪态

vm.setCurStatus(READY);

}

callScheduling();//请求分区调度

return VMK\_OK;

}

}

}

RK.6.5 T\_VMK\_ReturnCode vmkAckException(

T\_VOID

){

}

leftBufferSize\\剩余缓冲空间

RK.8.1 T\_VMK\_ReturnCode vmkEveWriteData(

T\_UDWORD\* dwTime,

T\_UWORD uwType,

T\_VOID\* vpData,

T\_UWORD uwDataLen

){

If(vmkStopVM ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

If(uwDataLen==0){

Return VMK\_INVALID\_SIZE;

}

If(notValidCheck(dwTime){

Return VMK\_INVALID\_ADDRESS;

}

If(notValidCheck(vpData){

Return VMK\_INVALID\_ADDRESS;

}

If(uwType<6){

Return VMK\_INVALID\_TYPE;

}

If(leftBufferSize< uwDataLen){

Return VMK\_INVALID\_SIZE;

}else{

writeDataInBuffer();

return VMK\_OK;

}

}

RK.9.1 T\_VMK\_ReturnCode vmkGetCPUTime(

T\_UDWORD \*udwpTime

){

If(notValidCheck(dwTime){// 如果<udwpTime>指向的地址空间没有被映射、地址空间不可读写访问

Return VMK\_INVALID\_ADDRESS;

}

If(successGetCurTime()){

Return VMK\_OK;

}

}

RK.9.2 T\_VMK\_ReturnCode vmkGetKernelInfo(

T\_VMK\_VMKBasicInfo \*tpBuf

){

If(notValidCheck(tpBuf){// 如果< tpBuf >指向的地址空间没有被映射、地址空间不可读写访问

Return VMK\_INVALID\_ADDRESS;

}

If(successGetKernelInfo()){

Return VMK\_OK;

}

}

RK.9.3 T\_VMK\_ReturnCode vmkGetSystemTicks(

T\_UWORD\* ticks

){

If(notValidCheck(ticks){// 如果< ticks >指向的地址空间没有被映射、地址空间不可读写访问

Return VMK\_INVALID\_ADDRESS;

}

If(successGetTicksInfo()){

Return VMK\_OK;

}

}

}

endOf APPVMADT

Machine KERVMADT{

appVM;

kerVM;

idlVM;

appVMLogicAddress;//应用分区的逻辑地址范围

canAccessInterfaceSet;// 分区所配特权接口集合

exceptionVMSet;//产生异常的分区集合

schedulingTable;//时间调度表

SSTWakenUpType;//时间调度表唤醒类型，包含基本、定时、时间窗或时间调度表启动唤醒类型

//获取应用分区的名字

RK.2.2 T\_VMK\_ReturnCode vmkInGetVMID(

T\_UBYTE \*ubpName,

T\_UWORD \*uwpVMID

){

If(ubpName==NULL){

uwpVMID=curVMID;

vm.<ID: uwpVMID >∈appVM∪kerVM;

if(vm==NULL){

return VMK\_INVALID\_NAME;

}

return VMK\_OK;

}

vm.<NAME:ubpName>∈appVM∪kerVM;

if(vm==NULL){

return VMK\_INVALID\_NAME;

}

\*uwpVMID=vm.<ID>;

return VMK\_OK;

}

//获取应用分区的名字

RK.2.5 T\_VMK\_ReturnCode vmkInGetVMName(

T\_UWORD uwVMID,

T\_UBYTE \*ubpName

){

If(uwVMID ==VMK\_NULL\_VID){

ubpName = curVM. < NAME >;

}

vm.<ID:uwVMID>∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

\*ubpName=copy(vm.<NAME>);

return VMK\_OK;

}

//获取产生异常的应用分区的ID号

RK.2.12 T\_VMK\_ReturnCode vmkInGetExceptionVM(

T\_UWORD \*uwVMID;

){

vm∈exceptionVMSet;

if(vm==NULL){

return VMK\_INVALID\_STATE;

}

If(vm.isDebugging()){

If(curVM∈ kerVM){

\*uwVMID= curVM.<ID>;

Return VMK\_OK;

}

Return VMK\_INVALID\_USER;

}else{

If(curVM.isNotManagePartition(vm)){

Return VMK\_INVALID\_USER;

}

}

If(calledInSuspend()){

Return VMK\_CALLED\_FROM\_ISR;

}

}

//获取应用分区的状态

RK.2.8 T\_VMK\_ReturnCode vmkInGetVMStatus(

T\_UWORD uwVMID,

T\_VMK\_VMStatus \*tpStatus

){

If(uwVMID ==VMK\_NULL\_VID){

If(curVM∈appVM){

\*tpStatus= curVM.<STATUS>;

~~return VMK\_OK;~~

}

};

vm.<ID:uwVMID>∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

\*tpStatus= vm.<STATUS>;

return VMK\_OK;

}

//获取系统中应用分区的个数

RK.2.19 T\_VMK\_ReturnCode vmkInGetVMCount(

T\_UWORD \*uwpVMCount

){

\*uwpVMCount=sizeOf(appVM);

If(\*uwpVMCount==NULL){

return VMK\_INVALID\_ADDRESS;

}

return VMK\_OK;

}

//启动核心分区或应用分区

RK.2.28 T\_VMK\_ReturnCode vmkInStartVM(

T\_UWORD uwVMID,

T\_VMK\_SchedType tType,

BOOL isSuspended

){

vm.<ID: uwVMID >∈appVM∪kerVM;

if(vm==NULL){// 如果<uwVMID>指定的分区不是应用分区或核心分区

return VMK\_INVALID\_ID;

}

If(type∉schedulingType){// 如果<tType>不表示默认调度类型、优先级调度类型或时间调度表调度类型

Return VMK\_INVALID\_TYPE;

}

If(vm.isOnStarting()){//如果<uwVMID>指定的分区已经被启动

Return VMK\_INVALID\_STATE;

}

vm.<ID: uwVMID >∈ kerVM;

if(vm!=NULL){ 启动核心分区

if(isSuspended==TRUE){ //启动就挂起的方式启动核心分区

return VMK\_INVALID\_ID;

}

If(tType== VMK\_SCHED\_TIME\_TABLE){// 如果欲用时间调度表调度类型启动核心分区，应返回VMK\_INVALID\_TYPE

Return VMK\_INVALID\_TYPE;

}

}

If(tType== VMK\_SCHED\_DEFAULT){// 如果调度类型为默认调度类型

if(isSuspended==TRUE){

vm.setCurStatus(VMK\_VM\_FIRSTRUN,HUNGUP);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和挂起态；

callScheduling(DEFAULT);// 采用用户所配置的调度类型或上一次运行的调度类型请求分区调度；

return VMK\_OK

}

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(DEFAULT);// 采用用户所配置的调度类型或上一次运行的调度类型请求分区调度；

return VMK\_OK

}

If(tType== VMK\_SCHED\_PRIORITY || tType== VMK\_SCHED\_TIME\_TABLE){// 如果调度类型为优先级调度类型或时间调度表调度类型

if(isSuspended==TRUE){

vm.setCurStatus(VMK\_VM\_FIRSTRUN,HUNGUP);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和挂起态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK

}

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//重启应用分区

RK.2.32 T\_VMK\_ReturnCode vmkInResetVM(

T\_UWORD uwVMID,

T\_VMK\_SchedType tType

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(type∉schedulingType){// 如果<tType>不表示默认调度类型、优先级调度类型或时间调度表调度类型

Return VMK\_INVALID\_TYPE;

}

if(vm.<STATUS:vmStatus>==unStarting){// 如果<uwVMId>指定的分区尚未启动

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>==READY){// 如果<uwVMID>指定的分区为就绪态

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==WAITING){// 如果<uwVMId>指定的分区的状态是等待态

vm.setCurWaitingTime(0);//无论<uwVMId>指定的分区的等待时间是否超时，均使其等待时间为0；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==HUNGUP){//如果<uwVMId>指定的分区处于挂起态

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.havaSecondException()){//如果<uwVMId>指定的分区产生了二次异常

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//停止应用分区

RK.2.30 T\_VMK\_ReturnCode vmkInStopVM(

T\_UWORD uwVMID

){

vm.<ID: uwVMID >∈ appVM;

if(vm!=NULL){// <uwVMId>指定的分区不是应用分区

return VMK\_INVALID\_ID;

}

if(vm.<STATUS:vmStatus>==unStarting){// 如果<uwVMId>指定的分区尚未启动

return VMK\_INVALID\_STATE；

}

if(vm.<STATUS:vmStatus>==READY){// 如果<uwVMId>指定的分区的状态为就绪态

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==WAITING){// 如果<uwVMId>指定的分区的状态是等待态

vm.setCurWaitingTime(0);//无论<uwVMId>指定的分区的等待时间是否超时，均使其等待时间为0；

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==HUNGUP){//如果<uwVMId>指定的分区处于挂起态

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.havaSecondException()){//如果<uwVMId>指定的分区产生了二次异常

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//挂起应用分区

RK.2.34 T\_VMK\_ReturnCode vmkInSuspendVM(

T\_UWORD uwVMID

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

if(vm.<STATUS:vmStatus>==unStarting || vm.<STATUS:vmStatus>==HUNGUP){// 如果<uwVMId>指定的分区尚未启动或者如果应用分区已经挂起

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>==READY){// 如果<uwVMID>指定的分区为就绪态

vm.setCurStatus(HUNGUP);//将<uwVMID>指定分区的状态置为挂起态；

callScheduling();//请求分区调度

return VMK\_OK;

}else{

vm.setCurStatus(HUNGUPandWAITING)；//设置应用分区挂起态标志位，如果应用分区已经是等待状态，则此时应用分区处于挂等态；

return VMK\_OK;

}

}

//解挂应用分区

RK.2.36 T\_VMK\_ReturnCode vmkInResumeVM(

T\_UWORD uwVMID

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(vm.isNotHUNGUP()){//应用分区尚未被挂起

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>== HUNGUPandWAITING){// 如果<uwVMID>指定的分区为挂等状态

Vm.clearCurStatus(HUNGUP); // 清除该应用分区的挂起状态

vm.setCurStatus(WAITING);// 清除该应用分区的挂起状态，使该应用分区处于等待状态；

return VMK\_OK;

}

If(vm.<STATUS:vmStatus>== HUNGUP){// 如果<uwVMID>指定的分区为挂起状态

vm.setCurStatus(READY);// 将其设置为就绪态；

callScheduling();//请求分区调度

return VMK\_OK;

}

}

//使当前核心分区放弃CPU，根据指定的唤醒类型来唤醒该分区

interruptSet;// 未处理的虚拟外部中断或虚拟服务中断集合

waitingQueue; //差分时间等待队列集合

readyQueue; //就绪队列；

schedulingTable

RK.2.21 T\_VMK\_ReturnCode vmkHaltVM(

T\_VMK\_WakeupType tWakeupType,

T\_UWORD uwTicks

){

If(curVM∈ idlVM){

Return VMK\_INVALID\_USER;

}

If(tWakeupType ==timeAwaken ){// 唤醒类型为定时唤醒类型时

If( watingTime==0){

Return VMK\_INVALID\_TIME;

}

clearCurVMStatus(READY);//清除当前分区的就绪状态

waitingQueue= waitingQueue∪<curVM, uwTicks >;//将当前分区插入到差分时间等待队列

setCurVMStatus(WAITING);// 设置当前分区状态为等待状态

setCurVMWakenUpType(tWakeupType);//设置当前分区的唤醒类型为<tWakeupType>

callScheduling();

return VMK\_OK;

}

If(tWakeupType==SSTAwakenType ){// 唤醒类型为时间调度表调度启动唤醒类型

If( curVM. TTSInfo<TTSType>!=PriorityScheduleType){// 当前分区的调度类型不是优先级调度

Return VMK\_INVALID\_TYPE;

}

readQueue= readQueue-<curVM>;//将当前分区从优先级就绪队列中移出；

setCurVMScheduling(TTSScheduleType);//设置当前分区调度类型为时间调度表调度；

setCurVMStatus(READY);//设置当前分区状态为就绪状态；

setCurVMWakenUpType(SSTAwakenType);//设置当前分区的唤醒类型为时间调度表启动唤醒类型；

callScheduling();//请求分区调度；

return VMK\_OK;

}

If(tWakeupType== timeWindowAwakenType){// 唤醒类型为时间窗唤醒类型

If( curVM. TTSInfo<TTSType>!= TTSScheduleType){// 当前分区的调度类型不是时间调度表调度

Return VMK\_INVALID\_TYPE;

}

giveupCPU();

setSchedulingTable(curVM,idlVM); //并设置时间调度表当前分区为IDLE分区；

recordWindowTime();//记录当前分区被停止的时间窗口

setCurVMWakenUpType(timeWindowAwakenType);//设置当前分区的唤醒类型为时间窗唤醒

callScheduling();//请求分区调度；

return VMK\_OK;

}

If(tWakeupType ==basicAwaken ){// 唤醒类型为基本唤醒类型时

If( watingTime==0){

clearCurVMStatus(READY);//清除当前分区的就绪状态

setCurVMStatus(WAITING);// 设置当前分区状态为等待状态

setCurVMWakenUpType(basicAwaken);//设置当前分区的唤醒类型为基本唤醒类型

callScheduling();

return VMK\_OK;

}

Return Return VMK\_INVALID\_TIME;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//根据应用分区的ID获取应用分区信息

RK.2.45 T\_VMK\_ReturnCode vmkInGetVMInfo(

T\_UWORD uwVMID,

T\_VMK\_VMBasicInfo \*tpBuf

){

if(uwVMID ==VMK\_NULL\_VID){// 如果<uwVMID>为VMK\_NULL\_VID，应获取当前分区的信息，返回VMK\_OK

\*tpBuf =curVMID;

return VMK\_OK;

}

//不为VMK\_NULL\_VID时

vm.<ID: uwVMID >∈appVM;

if(vm==NULL){// 如果不存在<uwVMID>指定的应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

\*tpBuf= vm.<vmINFO>;//如果存在<uwVMID>指定的应用分区，应获取该应用分区的信息，并返回VMK\_OK

return VMK\_OK;

}

PICset//虚拟中断号范围集合

PICHardwareset//实际的中断号范围集合

PICNotAllowed//当前分区屏蔽的外部中断集合

PICMapSet//中断映射表集合

RK.6.2 T\_VMK\_ReturnCode vmkDisablePIC(

T\_UWORD uwVIntNum

){

If(uwVIntNum∉PICset){// 如果<uwVIntNum>不在虚拟中断号范围内，应返回VMK\_INVALID\_INDEX

Return VMK\_INVALID\_INDEX;

}

Pic.<ID: uwVIntNum >∈PICNotAllowed;

If(pic!=NULL){// 如果虚拟中断号<uwVIntNum>对应的外部中断被当前分区屏蔽，应返回VMK\_MASKED

Return VMK\_MASKED;

}

picHardNum.<ID: uwVIntNum >∈PIDMapSet; 根据用户所配置的中断映射表将虚拟中断号<uwVIntNum>转换成硬件中断号

if(changeInValid()){//如果无法将虚拟中断号映射到硬件中断号或者转换后的硬件中断号大于等于硬件中断个数

return VMK\_INVALID\_STATE;

}

If(picHardNum ∉ PICHardwareset){// 如果用户所配置中断映射表中硬件中断号不在实际的中断号范围内

Return VMK\_INVALID\_INDEX;

}

If(disablePIDSuccessed()){//如果成功禁止当前分区的指定虚拟外部中断

Return VMK\_OK;

}

}

RK.6.3 T\_VMK\_ReturnCode vmkEnablePIC(

T\_UWORD uwVIntNum

){

If(uwVIntNum∉PICset){// 如果<uwVIntNum>不在虚拟中断号范围内，应返回VMK\_INVALID\_INDEX

Return VMK\_INVALID\_INDEX;

}

Pic.<ID: uwVIntNum >∈PIDNotAllowed;

If(pic!=NULL){// 如果虚拟中断号<uwVIntNum>对应的外部中断被当前分区屏蔽，应返回VMK\_MASKED

Return VMK\_MASKED;

}

picHardNum.<ID: uwVIntNum >∈PIDMapSet; 根据用户所配置的中断映射表将虚拟中断号<uwVIntNum>转换成硬件中断号

if(changeInValid()){//如果无法将虚拟中断号映射到硬件中断号或者转换后的硬件中断号大于等于硬件中断个数

return VMK\_INVALID\_STATE;

}

If(picHardNum ∉ PICHardwareset){// 如果用户所配置中断映射表中硬件中断号不在实际的中断号范围内

Return VMK\_INVALID\_INDEX;

}

If(enablePIDSuccessed()){//如果成功禁止当前分区的指定虚拟外部中断

Return VMK\_OK;

}

}

RK.6.6 T\_VMK\_ReturnCode vmkInInitializeVint(

T\_VMK\_VMVintEnMask \*tpVIntEnMask,

T\_VMK\_VINT\_HANDLER tpHandler

){

If(tpVIntEnMask==NULL || tpHandler==NULL){

Return VMK\_INVALID\_ADDRESS;

}

If(vm.tpInterruptIsNotInitialed()){//如果当前分区的虚拟中断尚未被初始化

Return VMK\_INVALID\_STATE;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

setPICOP(tpHandler, tpVIntEnMask,TRUE);// 根据<tpHandler>和<tpVIntEnMask>初始化当前分区的虚拟中断入口程序和屏蔽码，设置当前分区的虚拟中断初始化标识为TRUE，返回VMK\_OK

return VMK\_OK;

}

PICSet/虚拟中断集合

tickSet/tick中断集合

exceptionSet/异常集合

outerSet/外部中断集合

serviceSet/中断服务集合

RK.6.7 T\_VMK\_ReturnCode vmkGetVint(

T\_UWORD \*uwpVIntType,

T\_UWORD \*uwpVIntNum,

T\_UWORD \*uwpEntry

){

If(tpVIntEnMask==NULL || tpHandler==NULL || uwpEntry==NULL){

Return VMK\_INVALID\_ADDRESS;

}

If(vm.tpInterruptIsNotInitialed()){//如果当前分区的虚拟中断尚未被初始化

Return VMK\_INVALID\_STATE;

}

If(PICSet==NULL){// 如果没有虚拟中断(异常、tick中断、外部中断、服务请求中断)需要投递，应返回VMK\_FAIL

Return VMK\_FAIL;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

If(exceptionSet!=NULL){// 如果有异常需要投递

setUwpVIntType(VMK\_VINT\_EXCEPTION);

gainExceptionIDandAddress();//获取最高优先级的异常的中断号和入口程序地址

clearException();//清除该异常记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}else{

If(PICturing==0){// 如果没有异常需要投递并且全局虚拟中断开关为0

Return VMK\_FAIL;

}

}

If(exceptionSet!=NULL){// 如果有tick中断需要投递

setUwpVIntType(VMK\_VINT\_ TICK);

gainTickIDandAddress();//获取最高优先级的tick中断的中断号和入口程序地址

clearTick();//清除该异常记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}

If(outerSet!=NULL){// 如果有外部中断需要投递

setUwpVIntType(VMK\_VINT\_ EXINT);

gainOuterIDandAddress();//获取最高优先级的外部中断的中断号和入口程序地址

clearOuter();//清除该异常记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}

If(serviceSet!=NULL){// 如果有外部中断需要投递

setUwpVIntType(VMK\_VINT\_ SERVICE);

gainServiceIDandAddress();//获取最高优先级的外部服务请求中断的中断号和入口程序地址

clearService();//清除该请求中断的记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}

}

RK.6.8 T\_VMK\_ReturnCode vmkInSendServiceInt(

T\_UWORD uwVMID,

T\_UWORD uwServiceIntNum

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(uwVIntNum∉PICset){// 如果<uwVIntNum>不在虚拟中断号范围内，应返回VMK\_INVALID\_INDEX

Return VMK\_INVALID\_INDEX;

}

If(vm.isNotOnStarting()){

Return VMK\_INVALID\_STATE;

}

If(vm.tpInterruptIsNotInitialed()){//如果当前分区的虚拟中断尚未被初始化

Return VMK\_INVALID\_STATE;

}

If(vm.awakenType()!=VMK\_WAKEUP\_BASIC){// 定的分区的唤醒类型不是VMK\_WAKEUP\_BASIC

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

return VMK\_OK;

} else{

If(waitingTime==0){// 并且等待时间为0

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

awaken(vm);// 唤醒该分区

if(vm.isNotHungup()){//如果该分区没有被挂起，应设置该分区为就绪态

vm.setCurStatus(READY);

}

callScheduling();//请求分区调度

return VMK\_OK;

}

If(waitingTime<=ENDTIME){// 并且等待时间还未超时

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

waitingTimeQueue-vm;// 将该分区从差分时间等待队列上移除

awaken(vm);// 唤醒该分区

if(vm.isNotHungup()){//如果该分区没有被挂起，应设置该分区为就绪态

vm.setCurStatus(READY);

}

callScheduling();//请求分区调度

return VMK\_OK;

}

}

}

}

//END OF KERVMDAT

Machine IDLEVMADT{

appVM;

kerVM;

idlVM;

appVMLogicAddress;//应用分区的逻辑地址范围

canAccessInterfaceSet;// 分区所配特权接口集合

exceptionVMSet;//产生异常的分区集合

schedulingTable;//时间调度表

SSTWakenUpType;//时间调度表唤醒类型，包含基本、定时、时间窗或时间调度表启动唤醒类型

//获取应用分区的名字

RK.2.2 T\_VMK\_ReturnCode vmkInGetVMID(

T\_UBYTE \*ubpName,

T\_UWORD \*uwpVMID

){

If(ubpName==NULL){

uwpVMID=curVMID;

vm.<ID: uwpVMID >∈appVM∪kerVM;

if(vm==NULL){

return VMK\_INVALID\_NAME;

}

return VMK\_OK;

}

vm.<NAME:ubpName>∈appVM∪kerVM;

if(vm==NULL){

return VMK\_INVALID\_NAME;

}

\*uwpVMID=vm.<ID>;

return VMK\_OK;

}

//获取应用分区的名字

RK.2.5 T\_VMK\_ReturnCode vmkInGetVMName(

T\_UWORD uwVMID,

T\_UBYTE \*ubpName

){

If(uwVMID ==VMK\_NULL\_VID){

ubpName = curVM. < NAME >;

}

vm.<ID:uwVMID>∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

\*ubpName=copy(vm.<NAME>);

return VMK\_OK;

}

//获取产生异常的应用分区的ID号

RK.2.12 T\_VMK\_ReturnCode vmkInGetExceptionVM(

T\_UWORD \*uwVMID;

){

vm∈exceptionVMSet;

if(vm==NULL){

return VMK\_INVALID\_STATE;

}

If(vm.isDebugging()){

If(curVM∈ kerVM){

\*uwVMID= curVM.<ID>;

Return VMK\_OK;

}

Return VMK\_INVALID\_USER;

}else{

If(curVM.isNotManagePartition(vm)){

Return VMK\_INVALID\_USER;

}

}

If(calledInSuspend()){

Return VMK\_CALLED\_FROM\_ISR;

}

}

//获取应用分区的状态

RK.2.8 T\_VMK\_ReturnCode vmkInGetVMStatus(

T\_UWORD uwVMID,

T\_VMK\_VMStatus \*tpStatus

){

If(uwVMID ==VMK\_NULL\_VID){

If(curVM∈appVM){

\*tpStatus= curVM.<STATUS>;

~~return VMK\_OK;~~

}

};

vm.<ID:uwVMID>∈appVM;

if(vm==NULL){

return VMK\_INVALID\_ID;

}

\*tpStatus= vm.<STATUS>;

return VMK\_OK;

}

//获取系统中应用分区的个数

RK.2.19 T\_VMK\_ReturnCode vmkInGetVMCount(

T\_UWORD \*uwpVMCount

){

\*uwpVMCount=sizeOf(appVM);

If(\*uwpVMCount==NULL){

return VMK\_INVALID\_ADDRESS;

}

return VMK\_OK;

}

//启动核心分区或应用分区

RK.2.28 T\_VMK\_ReturnCode vmkInStartVM(

T\_UWORD uwVMID,

T\_VMK\_SchedType tType,

BOOL isSuspended

){

vm.<ID: uwVMID >∈appVM∪kerVM;

if(vm==NULL){// 如果<uwVMID>指定的分区不是应用分区或核心分区

return VMK\_INVALID\_ID;

}

If(type∉schedulingType){// 如果<tType>不表示默认调度类型、优先级调度类型或时间调度表调度类型

Return VMK\_INVALID\_TYPE;

}

If(vm.isOnStarting()){//如果<uwVMID>指定的分区已经被启动

Return VMK\_INVALID\_STATE;

}

vm.<ID: uwVMID >∈ kerVM;

if(vm!=NULL){ 启动核心分区

if(isSuspended==TRUE){ //启动就挂起的方式启动核心分区

return VMK\_INVALID\_ID;

}

If(tType== VMK\_SCHED\_TIME\_TABLE){// 如果欲用时间调度表调度类型启动核心分区，应返回VMK\_INVALID\_TYPE

Return VMK\_INVALID\_TYPE;

}

}

If(tType== VMK\_SCHED\_DEFAULT){// 如果调度类型为默认调度类型

if(isSuspended==TRUE){

vm.setCurStatus(VMK\_VM\_FIRSTRUN,HUNGUP);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和挂起态；

callScheduling(DEFAULT);// 采用用户所配置的调度类型或上一次运行的调度类型请求分区调度；

return VMK\_OK

}

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(DEFAULT);// 采用用户所配置的调度类型或上一次运行的调度类型请求分区调度；

return VMK\_OK

}

If(tType== VMK\_SCHED\_PRIORITY || tType== VMK\_SCHED\_TIME\_TABLE){// 如果调度类型为优先级调度类型或时间调度表调度类型

if(isSuspended==TRUE){

vm.setCurStatus(VMK\_VM\_FIRSTRUN,HUNGUP);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和挂起态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK

}

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//重启应用分区

RK.2.32 T\_VMK\_ReturnCode vmkInResetVM(

T\_UWORD uwVMID,

T\_VMK\_SchedType tType

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(type∉schedulingType){// 如果<tType>不表示默认调度类型、优先级调度类型或时间调度表调度类型

Return VMK\_INVALID\_TYPE;

}

if(vm.<STATUS:vmStatus>==unStarting){// 如果<uwVMId>指定的分区尚未启动

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>==READY){// 如果<uwVMID>指定的分区为就绪态

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==WAITING){// 如果<uwVMId>指定的分区的状态是等待态

vm.setCurWaitingTime(0);//无论<uwVMId>指定的分区的等待时间是否超时，均使其等待时间为0；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==HUNGUP){//如果<uwVMId>指定的分区处于挂起态

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

if(vm.havaSecondException()){//如果<uwVMId>指定的分区产生了二次异常

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

vm.setCurStatus(VMK\_VM\_FIRSTRUN,READY);//将<uwVMID>指定分区的状态置为VMK\_VM\_FIRSTRUN和就绪态；

callScheduling(tType);// 根据调度类型<tType>请求分区调度

return VMK\_OK;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//停止应用分区

RK.2.30 T\_VMK\_ReturnCode vmkInStopVM(

T\_UWORD uwVMID

){

vm.<ID: uwVMID >∈ appVM;

if(vm!=NULL){// <uwVMId>指定的分区不是应用分区

return VMK\_INVALID\_ID;

}

if(vm.<STATUS:vmStatus>==unStarting){// 如果<uwVMId>指定的分区尚未启动

return VMK\_INVALID\_STATE；

}

if(vm.<STATUS:vmStatus>==READY){// 如果<uwVMId>指定的分区的状态为就绪态

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==WAITING){// 如果<uwVMId>指定的分区的状态是等待态

vm.setCurWaitingTime(0);//无论<uwVMId>指定的分区的等待时间是否超时，均使其等待时间为0；

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.<STATUS:vmStatus>==HUNGUP){//如果<uwVMId>指定的分区处于挂起态

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

if(vm.havaSecondException()){//如果<uwVMId>指定的分区产生了二次异常

vm.setCurStatus(SLEEPING);//设置<uwVMId>指定的分区为休眠态；

vm.restoreBackToInitialStatus();//恢复<uwVMID>指定的分区的运行时的数据到初始化时的状态；

callScheduling();//请求分区调度

return VMK\_OK;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//挂起应用分区

RK.2.34 T\_VMK\_ReturnCode vmkInSuspendVM(

T\_UWORD uwVMID

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

if(vm.<STATUS:vmStatus>==unStarting || vm.<STATUS:vmStatus>==HUNGUP){// 如果<uwVMId>指定的分区尚未启动或者如果应用分区已经挂起

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>==READY){// 如果<uwVMID>指定的分区为就绪态

vm.setCurStatus(HUNGUP);//将<uwVMID>指定分区的状态置为挂起态；

callScheduling();//请求分区调度

return VMK\_OK;

}else{

vm.setCurStatus(HUNGUPandWAITING)；//设置应用分区挂起态标志位，如果应用分区已经是等待状态，则此时应用分区处于挂等态；

return VMK\_OK;

}

}

//解挂应用分区

RK.2.36 T\_VMK\_ReturnCode vmkInResumeVM(

T\_UWORD uwVMID

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(vm.isNotHUNGUP()){//应用分区尚未被挂起

return VMK\_INVALID\_STATE；

}

If(vm.<STATUS:vmStatus>== HUNGUPandWAITING){// 如果<uwVMID>指定的分区为挂等状态

Vm.clearCurStatus(HUNGUP); // 清除该应用分区的挂起状态

vm.setCurStatus(WAITING);// 清除该应用分区的挂起状态，使该应用分区处于等待状态；

return VMK\_OK;

}

If(vm.<STATUS:vmStatus>== HUNGUP){// 如果<uwVMID>指定的分区为挂起状态

vm.setCurStatus(READY);// 将其设置为就绪态；

callScheduling();//请求分区调度

return VMK\_OK;

}

}

//使当前核心分区放弃CPU，根据指定的唤醒类型来唤醒该分区

interruptSet;// 未处理的虚拟外部中断或虚拟服务中断集合

waitingQueue; //差分时间等待队列集合

readyQueue; //就绪队列；

schedulingTable

RK.2.21 T\_VMK\_ReturnCode vmkHaltVM(

T\_VMK\_WakeupType tWakeupType,

T\_UWORD uwTicks

){

If(curVM∈ idlVM){

Return VMK\_INVALID\_USER;

}

If(tWakeupType ==timeAwaken ){// 唤醒类型为定时唤醒类型时

If( watingTime==0){

Return VMK\_INVALID\_TIME;

}

clearCurVMStatus(READY);//清除当前分区的就绪状态

waitingQueue= waitingQueue∪<curVM, uwTicks >;//将当前分区插入到差分时间等待队列

setCurVMStatus(WAITING);// 设置当前分区状态为等待状态

setCurVMWakenUpType(tWakeupType);//设置当前分区的唤醒类型为<tWakeupType>

callScheduling();

return VMK\_OK;

}

If(tWakeupType==SSTAwakenType ){// 唤醒类型为时间调度表调度启动唤醒类型

If( curVM. TTSInfo<TTSType>!=PriorityScheduleType){// 当前分区的调度类型不是优先级调度

Return VMK\_INVALID\_TYPE;

}

readQueue= readQueue-<curVM>;//将当前分区从优先级就绪队列中移出；

setCurVMScheduling(TTSScheduleType);//设置当前分区调度类型为时间调度表调度；

setCurVMStatus(READY);//设置当前分区状态为就绪状态；

setCurVMWakenUpType(SSTAwakenType);//设置当前分区的唤醒类型为时间调度表启动唤醒类型；

callScheduling();//请求分区调度；

return VMK\_OK;

}

If(tWakeupType== timeWindowAwakenType){// 唤醒类型为时间窗唤醒类型

If( curVM. TTSInfo<TTSType>!= TTSScheduleType){// 当前分区的调度类型不是时间调度表调度

Return VMK\_INVALID\_TYPE;

}

giveupCPU();

setSchedulingTable(curVM,idlVM); //并设置时间调度表当前分区为IDLE分区；

recordWindowTime();//记录当前分区被停止的时间窗口

setCurVMWakenUpType(timeWindowAwakenType);//设置当前分区的唤醒类型为时间窗唤醒

callScheduling();//请求分区调度；

return VMK\_OK;

}

If(tWakeupType ==basicAwaken ){// 唤醒类型为基本唤醒类型时

If( watingTime==0){

clearCurVMStatus(READY);//清除当前分区的就绪状态

setCurVMStatus(WAITING);// 设置当前分区状态为等待状态

setCurVMWakenUpType(basicAwaken);//设置当前分区的唤醒类型为基本唤醒类型

callScheduling();

return VMK\_OK;

}

Return Return VMK\_INVALID\_TIME;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

}

//根据应用分区的ID获取应用分区信息

RK.2.45 T\_VMK\_ReturnCode vmkInGetVMInfo(

T\_UWORD uwVMID,

T\_VMK\_VMBasicInfo \*tpBuf

){

if(uwVMID ==VMK\_NULL\_VID){// 如果<uwVMID>为VMK\_NULL\_VID，应获取当前分区的信息，返回VMK\_OK

\*tpBuf =curVMID;

return VMK\_OK;

}

//不为VMK\_NULL\_VID时

vm.<ID: uwVMID >∈appVM;

if(vm==NULL){// 如果不存在<uwVMID>指定的应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

\*tpBuf= vm.<vmINFO>;//如果存在<uwVMID>指定的应用分区，应获取该应用分区的信息，并返回VMK\_OK

return VMK\_OK;

}

}

RK.2.45 T\_VMK\_ReturnCode vmkInGetVMInfo(

T\_UWORD uwVMID,

T\_VMK\_VMBasicInfo \*tpBuf

){

if(uwVMID ==VMK\_NULL\_VID){// 如果<uwVMID>为VMK\_NULL\_VID，应获取当前分区的信息，返回VMK\_OK

\*tpBuf =curVMID;

return VMK\_OK;

}

//不为VMK\_NULL\_VID时

vm.<ID: uwVMID >∈appVM;

if(vm==NULL){// 如果不存在<uwVMID>指定的应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

\*tpBuf= vm.<vmINFO>;//如果存在<uwVMID>指定的应用分区，应获取该应用分区的信息，并返回VMK\_OK

return VMK\_OK;

}

RK.6.6 T\_VMK\_ReturnCode vmkInInitializeVint(

T\_VMK\_VMVintEnMask \*tpVIntEnMask,

T\_VMK\_VINT\_HANDLER tpHandler

){

If(tpVIntEnMask==NULL || tpHandler==NULL){

Return VMK\_INVALID\_ADDRESS;

}

If(vm.tpInterruptIsNotInitialed()){//如果当前分区的虚拟中断尚未被初始化

Return VMK\_INVALID\_STATE;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

setPICOP(tpHandler, tpVIntEnMask,TRUE);// 根据<tpHandler>和<tpVIntEnMask>初始化当前分区的虚拟中断入口程序和屏蔽码，设置当前分区的虚拟中断初始化标识为TRUE，返回VMK\_OK

return VMK\_OK;

}

PICSet/虚拟中断集合

tickSet/tick中断集合

exceptionSet/异常集合

outerSet/外部中断集合

serviceSet/中断服务集合

RK.6.7 T\_VMK\_ReturnCode vmkGetVint(

T\_UWORD \*uwpVIntType,

T\_UWORD \*uwpVIntNum,

T\_UWORD \*uwpEntry

){

If(tpVIntEnMask==NULL || tpHandler==NULL || uwpEntry==NULL){

Return VMK\_INVALID\_ADDRESS;

}

If(vm.tpInterruptIsNotInitialed()){//如果当前分区的虚拟中断尚未被初始化

Return VMK\_INVALID\_STATE;

}

If(PICSet==NULL){// 如果没有虚拟中断(异常、tick中断、外部中断、服务请求中断)需要投递，应返回VMK\_FAIL

Return VMK\_FAIL;

}

If(calledInSuspend()){//如果在中断处理程序中执行此接口，应返回VMK\_CALLED\_FROM\_ISR。

Return VMK\_CALLED\_FROM\_ISR;

}

If(exceptionSet!=NULL){// 如果有异常需要投递

setUwpVIntType(VMK\_VINT\_EXCEPTION);

gainExceptionIDandAddress();//获取最高优先级的异常的中断号和入口程序地址

clearException();//清除该异常记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}else{

If(PICturing==0){// 如果没有异常需要投递并且全局虚拟中断开关为0

Return VMK\_FAIL;

}

}

If(exceptionSet!=NULL){// 如果有tick中断需要投递

setUwpVIntType(VMK\_VINT\_ TICK);

gainTickIDandAddress();//获取最高优先级的tick中断的中断号和入口程序地址

clearTick();//清除该异常记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}

If(outerSet!=NULL){// 如果有外部中断需要投递

setUwpVIntType(VMK\_VINT\_ EXINT);

gainOuterIDandAddress();//获取最高优先级的外部中断的中断号和入口程序地址

clearOuter();//清除该异常记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}

If(serviceSet!=NULL){// 如果有外部中断需要投递

setUwpVIntType(VMK\_VINT\_ SERVICE);

gainServiceIDandAddress();//获取最高优先级的外部服务请求中断的中断号和入口程序地址

clearService();//清除该请求中断的记录

turnoffPIC();//关闭全局虚拟中断开关

return VMK\_OK;

}

}

RK.6.8 T\_VMK\_ReturnCode vmkInSendServiceInt(

T\_UWORD uwVMID,

T\_UWORD uwServiceIntNum

){

vm.<ID: uwpVMID >∈appVM;

if(vm==NULL){// 如果<uwVMID>表示的分区不是应用分区，应返回VMK\_INVALID\_ID

return VMK\_INVALID\_ID;

}

If(uwVIntNum∉PICset){// 如果<uwVIntNum>不在虚拟中断号范围内，应返回VMK\_INVALID\_INDEX

Return VMK\_INVALID\_INDEX;

}

If(vm.isNotOnStarting()){

Return VMK\_INVALID\_STATE;

}

If(vm.tpInterruptIsNotInitialed()){//如果当前分区的虚拟中断尚未被初始化

Return VMK\_INVALID\_STATE;

}

If(vm.awakenType()!=VMK\_WAKEUP\_BASIC){// 定的分区的唤醒类型不是VMK\_WAKEUP\_BASIC

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

return VMK\_OK;

} else{

If(waitingTime==0){// 并且等待时间为0

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

awaken(vm);// 唤醒该分区

if(vm.isNotHungup()){//如果该分区没有被挂起，应设置该分区为就绪态

vm.setCurStatus(READY);

}

callScheduling();//请求分区调度

return VMK\_OK;

}

If(waitingTime<=ENDTIME){// 并且等待时间还未超时

sendPIC(vm);// 向<uwVMID>指定的分区发送服务请求中断

waitingTimeQueue-vm;// 将该分区从差分时间等待队列上移除

awaken(vm);// 唤醒该分区

if(vm.isNotHungup()){//如果该分区没有被挂起，应设置该分区为就绪态

vm.setCurStatus(READY);

}

callScheduling();//请求分区调度

return VMK\_OK;

}

}

}

//END OF IDLEVMDAT

RK.3.1 T\_VMK\_ReturnCode vmkStartSysWatchdog(

T\_VOID

){

If(vmkStartSysWatchdog ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

If(watchDogIsNotConfiged()){//如果用户未配置系统看门狗，应返回VMK\_INVALID\_ADDRESS

Return VMK\_INVALID\_ADDRESS;

}

if(watchDogIsSuccessed()){//如果启动系统看门狗成功，应返回VMK\_OK

return VMK\_OK;

}else{

Return VMK\_FAIL;

}

}

RK.3.2 T\_VMK\_ReturnCode vmkFeedSysWatchdog(

T\_VOID

){

If(vmkFeedSysWatchdog ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

If(watchDogIsNotConfiged()){//如果用户未配置系统看门狗，应返回VMK\_INVALID\_ADDRESS

Return VMK\_INVALID\_ADDRESS;

}

If(feedWatchDogSuccessed()){//如果进行系统看门狗喂狗成功，应返回VMK\_OK

Return VMK\_OK;

}else{//如果进行系统看门狗喂狗失败，应返回VMK\_FAIL

Return VMK\_FAIL;

}

}

RK.3.3 T\_VMK\_ReturnCode vmkStopSysWatchdog(

T\_VOID

){

If(vmkStopSysWatchdog ∉ canAccessInterfaceSet){// 当前分区是否有权限调用本接口

Return VMK\_INVALID\_USER;

}

If(watchDogIsNotConfiged()){//如果用户未配置系统看门狗，应返回VMK\_INVALID\_ADDRESS

Return VMK\_INVALID\_ADDRESS;

}

If(stopWatchDogSuccessed()){//如果停止系统看门狗成功，应返回VMK\_OK

Return VMK\_OK;

}else{//如果停止系统看门狗失败，应返回VMK\_FAIL

Return VMK\_FAIL;

}

}

RK.4.1 T\_VMK\_ReturnCode vmkAckVMWatchdogHook(

T\_VOID

){

If(curVM. watchDogIsNotConfiged()){//如果当前应用分区没有配置看门狗，返回VMK\_FAIL

Return VMK\_FAIL;

}

If(curVM.watchDogIsCalled()){//如果当前应用分区的看门狗回调函数正在被执行，返回VMK\_OK

Return VMK\_OK;

}else{//如果当前应用分区的看门狗未处于回调函数执行状态，返回VMK\_INVALID\_STATE

Return VMK\_INVALID\_STATE;

}

}

RK.4.2 T\_VMK\_ReturnCode vmkFeedVMWatchdog(

T\_VOID

){

If(curVM. watchDogIsNotConfiged()){//如果当前应用分区没有配置看门狗，返回VMK\_FAIL

Return VMK\_FAIL;

}

If(curVM. watchDogWorkedError ()){//如果当前应用分区的看门狗未处于正常工作状态

Return VMK\_INVALID\_STATE;

}

If(feedWatchDogSuccessed()){//如果进行系统看门狗喂狗成功，应返回VMK\_OK

Return VMK\_OK;

}

}

RK.4.3 T\_VMK\_ReturnCode vmkStartVMWatchdog(

T\_VOID

){

If(curVM. watchDogIsNotConfiged()){//如果当前应用分区没有配置看门狗，返回VMK\_FAIL

Return VMK\_FAIL;

}

If(curVM. watchDogIsNotInitialed() || curVM.watchDogIsStoped()){//如果当前应用分区的看门狗未处于初始化状态或停止状态

Return VMK\_INVALID\_STATE;

}

if(watchDogIsSuccessed()){//如果启动系统看门狗成功，应返回VMK\_OK

return VMK\_OK;

}

If(curVM. feedDogTime())>=0){// 如果当前应用分区看门狗最大喂狗时间等于0

Return VMK\_INVALID\_TIME;

}

}

RK.4.4 T\_VMK\_ReturnCode vmkStopVMWatchdog(

T\_VOID

){

If(curVM. watchDogIsNotConfiged()){//如果当前应用分区没有配置看门狗，返回VMK\_FAIL

Return VMK\_FAIL;

}

If(curVM. watchDogIsNotInitialed() || curVM.watchDogIsStoped()){//如果当前应用分区的看门狗未处于初始化状态或停止状态

Return VMK\_INVALID\_STATE;

}

If(stopWatchDogSuccessed()){//如果停止系统看门狗成功，应返回VMK\_OK

Return VMK\_OK;

}

}

A0=ω(vm'0, uwpVMID'0){1RK.2.1.T\_VMK\_ReturnCode vmkGetVMID(

T\_UBYTE \*ubpName,

T\_UWORD \*uwpVMID})}

A1=ω(vm'0, uwpVMID'0){

If(ubpName==NULL){

\*uwpVMID=curVM<ID>;

return VMK\_OK;}}{Q1}

A2= ubpName==NULL→ω(vm'0, uwpVMID'0){ uwpVMID=curVM<ID>;

return VMK\_OK;}{Q1}

A3=ω(vm'0, uwpVMID'0){ uwpVMID=curVM<ID>;return VMK\_OK;}

→ uwpVMID'1== curVM<ID>∧vmkGetVMID( )= VMK\_OK

A4= ubpName==NULL→ uwpVMID'1== curVM<ID>∧vmkGetVMID( )= VMK\_OK

A5= ubpName!=NULL→ω(vm'0, uwpVMID'0){Q1}

A6=ω(vm'0, uwpVMID'0){Q1}

→ω(vm'0){if(NoValidName(ubpName)){

return VMK\_INVALID\_ADDRESS;}}{Q2}

A7= ω(vm'0, uwpVMID'0){if(NoValidName(ubpName)){return VMK\_INVALID\_ADDRESS;}}{Q2}

→( (NoValidName(ubpName) →ω(vm'0, uwpVMID'0){ return VMK\_INVALID\_ADDRESS;})

A8= ubpName!=NULL

→( (NoValidName(ubpName) →ω(vm'0, uwpVMID'0){ return VMK\_INVALID\_ADDRESS;})

A9=ω(vm'0, uwpVMID'0){ return VMK\_INVALID\_ADDRESS;})

→ vmkGetVMID()=VMK\_INVALID\_ADDRESS

A10= ubpName!=NULL∧NoValidName(ubpName) → vmkGetVMID()=VMK\_INVALID\_ADDRESS

A11= ω(vm'0, uwpVMID'0){if(NoValidName(ubpName)){return VMK\_INVALID\_ADDRESS;}}{Q2}

→( !(NoValidName(ubpName) →ω(vm'0, uwpVMID'0){Q2})

A12= ubpName!=NULL∧ !(NoValidName(ubpName)

→ω(vm'0, uwpVMID'0){Q2})

A13= ubpName!=NULL∧ !(NoValidName(ubpName)

→ω(vm'0, uwpVMID'0){ if(vm.<NAME:ubpName>∈appVM){

return VMK\_INVALID\_NAME;}}){Q3}

A14=ω(vm'0, uwpVMID'0){ if(vm.<NAME:ubpName>∉appVM){

return VMK\_INVALID\_NAME;}}){Q3}

→ vm'1.<NAME:ubpName>∉appVM→ω(vm'1, uwpVMID'0){ return VMK\_INVALID\_NAME }

A15=ω(vm'1, uwpVMID'0){ return VMK\_INVALID\_NAME }

→ vmkGetVMID= VMK\_INVALID\_NAME

A16= ubpName!=NULL∧ !(NoValidName(ubpName) ∧ vm'1.<NAME:ubpName>∉appVM

→ vmkGetVMID= VMK\_INVALID\_NAME

A17=ω(vm'0, uwpVMID'0){ if(vm.<NAME:ubpName>∉appVM){

return VMK\_INVALID\_NAME;}}){Q3}

→ vm'1.<NAME:ubpName>∈appVM→ω(vm'1, uwpVMID'0){ Q3}

A18= ubpName!=NULL∧!(NoValidName(ubpName)

→ vm'1.<NAME:ubpName>∈appVM→ω(vm'1, uwpVMID'0){ Q3}

A19=ω(vm'1, uwpVMID'0){Q3}

→\*uwpVMID==vm.<ID>∧ω(vm'1, uwpVMID'1){return VMK\_OK;}

A20=ω(vm'1, uwpVMID'1){return VMK\_OK;}

→ vmkGetVMID( )= VMK\_OK

A21= ubpName!=NULL∧!(NoValidName(ubpName)∧ vm'1.<NAME:ubpName>∈appVM

→\* uwpVMID'1==vm.<ID>∧ vmkGetVMID( )= VMK\_OK

A4= ubpName==NULL→ \*uwpVMID'1== curVM<ID>∧vmkGetVMID( )= VMK\_OK

A10= ubpName!=NULL∧NoValidName(ubpName) → vmkGetVMID()=VMK\_INVALID\_ADDRESS

A16= ubpName!=NULL∧!(NoValidName(ubpName) ∧vm'1.<NAME:ubpName>∉appVM

→ vmkGetVMID= VMK\_INVALID\_NAME

∀∃∧∨¬→↔⊕⇔⇒≠╞├⊂⊆⊄∈∉∅φ∩∪∪∩Γ≈≡≥≤⊙∴∴∵∷∈≠≡≌±÷×⊥∞∘•◁αβωε'

Α α [alpha]

Β β [beta]

Γ γ [gamma]

Δ δ [delta]

Ε ε [epsilon]

Ζ ζ [zeta]

Η η [eta]

Θ θ [theta]

Ι ι [iota]

Κ κ [kappa]

Λ λ [lamda]

Μ μ [mu]

Ν ν [nu]

Ξ ξ [xi ksai]

Ο ο [omicron]

Π π [pi]

Ρ ρ [rho rou]

Σ σ ς [sigma]

Τ τ [tau]

Υ υ [upsilon]

Φ φ  [phi]

Χ χ [chi kai]

Ψ ψ [psi psai]

Ω ω [omega]