TasksHandleList

进入Simulation

TaskHandleList初始化在main.cpp中通过TimerStart()之后，在`UFS_TasksFsmInit(gUTTaskMngArry);`中`xxxFsmInit();`初始化了一系列的FSM(状态机)、`UFS_TaskMngInit();`初始化了`gTaskMngArry`，数组中包含一些列的FsmProcess和TimerSchedule。

之后便是一些相关API和Boot启动。

而后，通过windows创建一个主机线程GetInfo(...) 在GetInfo中，CaseCnt计数主机任务数量。通过CaseRegister(...)建立对应的Host任务

- 在本例流程中，建立了一个IO_HOST_WRITE Sequence用于追踪。

当进入整个simulation任务流程前，将SLOTDOWN置1。

simulation任务在一个while(1)当中，当且晋档超时`timeUpFlag == TRUE`跳出simulation任务(此时是否已经完成simulation? 从当前的调试情况来看，确实是这样。当simulation完成后才会有timeUpFlag=TRUE的情况)

Simulation流程

在第一个while(1)中，需`SLOTDOWN == 0`才可以退出。当且仅当HostProcess发送IO后，SLOTDOWN才会置0退出。

然后进入第一个for(...)中，进行一些Que LBA 等等的配置。紧接着while(1)进入TaskSchedule调度等待其运行完成。

然后进入第二个for(...)，同样进行一些Que之类的配置。

之后进入while(1)，进入Write流程的调度之中。

Write流程

进入TaskSchedule()->进入TasksHandleList通过遍历g_taskIndexHeadArry链表找到gTaskMngArry，然后遍历进入gTaskMngArry中的函数(包含UFS_TasksFsmInit中初始化的全部Process)中去。
`gTaskMngArry[index].handler();`

遍历顺序如下：

1. TimerSchedule
2. GetNewCmdProcess
3. Write_FsmProcess
4. Cache_FsmProcess
5. WriteResp_FsmProcess
6. ReadFsmProcess
7. ReadRespFsmProcess
8. RPMBFsmProcess
9. ReqFsmProcess

10. FFlushFsmProcess
11. BeNormalTaskProccess
12. BeBgTaskProcess
13. BeAdminTaskProcess
14. EvtTaskProcess
15. FTL_DisPatchCacheReq
16. FTL_DisPatchNandReqDone

FTL_DisPatchCacheReq 流程

FTL_DisPatchCacheReq将会处理三种类型的队列，包括：g_FeAdminReqQueue, g_FeReadReqQueue, g_FeWriteReqQueue。此三种队列均从FE端进行打包。

- 处理Write流程

---

g_FeWriteReqQueue首先被取出指向CurQueue

判空`OS_Queue_IsEmpty` 若空则进行下一轮处理队列的流程

非空， 则进入whlie(队列非空)流程处理中：

首先， 使用Queue_PeekHead获取ReqId并由此从g_FeReqArry中取出Request Node(结点)放入ReqNode

然后， 进入`FtlWriteCmdProc`(FTL Write Commond Process)中， 将ReqNode传入`FtlSubmitCacheWrite`

进入`FtlSubmitCacheWrite`后立即通过`LuGet(USER_LU_ID)`返回`g_LogicUnits(USER_LU_ID)`(LogicUnit_t)的地址，并以此创建一个`*UserLu`(UserLu_t)指向该片地址。

然后，再创建一个`*CurCtx(CurWrCtx_t)`指向`UserLu->CurWrCtx`。

此时，创建了一个地址指针UserLu指向地址指针数组g_LogicUnits[0]的地址。

如果FtlBoot没有启动成功， 则 返回`FTL_ERR_PENDING`的state退出当前流程。

如果UserLu中存在有FeNode即`CurCtx->FeNode != NULL`且该Node等于当前传入FeNode 则 sys_assert 否则令其指向当前传入的FeReqNode然后处理当前FeReqNode

判当前指向FeNode是否为空 sys_assert判断并置其上下文ForceFlag为TRUE

置上下文ErrCode为FTL_ERR_OK

进入`FsmCtxRun`

- 处理`_fsm_ctx_run`流程

根据Fe_DisPatchCacheReq传来的ctx中，将进行`UserWriteCtxCheck`

从ctx中找到并进入(相应的处理函数，这里是)UserWriteCtxCheck并返回其值`fsmres`

根据处理结果返回值fsmres 进行FSM的状态切换

- 处理UserWriteCtxCheck

进入UserWriteCtxCheck后首先进行了一次容器转换ContainerOf(...)

然后创建一个局部*user指针指向UserLu，也就是g_LogicUnits这片地址。

返回一个IO是否Blocked的标志给Blockflags

当且仅当未堵塞、USER_LU_F_HOST_PAUSE未发生、User当前Aspb没有被强制结束

则 返回 FSM_CONT 状态机正常进行计数

否则 返回 FSM_QUIT 状态机放弃当前任务，同时UserLu的上下文应置为FTL_ERR_PENDING(挂起)

- 通过_fsm_ctx_run进入UserWriteCtxSubmitReq

进入UserWriteCtxSubmitReq后首先进行了一次容器转换ContainerOf(...)返回当前fsm的ctx

然后创建一个*Node指针指向当前ctx的FeNode

创建一个局部变量*FtlReq指针指向当前ctx的req

当FeNode的Req类型是Write时：

使用SglGetSize(&CurCCtx->req->sgl)计算Datasize

如果DataSize大于UserDu的空间 则 置Flags FTL_REQ_F_SGL 需求SGL Opcode=FTL_OP_WRITE

通过FtlReqSetup和赋值 设置FtlReq

进入FtlSubmit(FtlReq)返回g_LogicUnits的submit返回值

即将该LU(UserLu)和FtlReq(write)传入其LogicUnitMethod(为submit方法：submit a IO to logical space)

在该LU和Write op下的submit方法指向 UserLuSubmit 即要求UserLuSubmit的返回值

同样地，要求其返回值

则根据LU(UserLu)和FtlReq(Write) 所以进入了g_UserSubmitOpHandlers(包含Read Write Trim Flush Gc 5种方法)的Write中

传入后，进入UserWrite流程

- 返回FTL_DisPatchCacheReq

对g_FeNodeOp进行对应操作之后，根据返回值

如果出现OOM或者Blocked则return

如果出现挂起 则break

其他则根据ReqId将当前的队列Queue_Dequeue

FTL_DisPatchNandReqDone 流程

UserWrite 流程

进入Userwrite

立即将LU的地址转换成`UserLu_t* user`进行解析，其中传入的pReq为FtlReq_t。创建`FtlIO_Ctx_t *hostCtx`(Ftl主机的IO上下文)，设置User页分配类型为`USER_HOST_PAGE_ALLOC`，创建`reqCnt = req->u8ReqCnt`(req计数变量)，创建`duCnts`(DU计数变量：通过reqCnt进行转换)，创建Page_ordered，设置user的PageALoc方法，创建`RlutLu_t *ru`，进入Step 1。

- Step 1

进行LuMakeDirty判断，如果LuMakeDirtyDone则`LuFlags &= ~LU_F_CLEAN`即LuMakeDirty返回真，跳出第一个Step 1进入Step 2。

`FtlReq->nandReq == NULL`则`FtlGetNandReq`

执行过`FtlGetNandReq`后`FtlReq->nandReq == NULL`仍成立，则`NandReqPendLu`挂起(Nand LU) 并返回FTL_ERR_OK

如果FtlReq是Req Gc 则返回`UserGcWrite`

- Step 2

判： 如果Logical Space缺乏可用资源(LU 的IO Blocked Flags != 0)，则置Flags为FTL_REQ_F_BLKING且pend并插入ftl req queue返回FTL_ERR_OK。

然后通过`FtlGetIoCtxById`(g_ftlHostCtxs)返回给主机上下文变量(hostCtx)。

判： g_powerLoss(掉电标志)假且`FC_Check`假 则`LuIoBlkBy`且goto `pend`

pend:

置Flags为FTL_REQ_F_BLKING且pend并插入ftl req queue返回FTL_ERR_OK。

- Step 3

将user中的rlutLu提出来放入ru

判： RlutEntryReserve返回假，则`LuIoBlkBy`(LU_IO_BLK_F_RLUT)且goto `got_return`

- Step 4

返回`PageAllocatorOrder`给page_ordered

判： page_ordered为零，则`LuIoBlkBy`(LU_IO_BLK_F_PAGE)并`RlutRetun`(ru->u16Validcnt += duCnts)

gotoreturn:

若PageAllocatorFull返回真(page分配器已满)则`UserConverTrigger`(启动User Convert)并goto`pend`。

将主机上下文(hostCtx)、req、pga、ru、reqCnt传入`UserWriteCtxSubmit`

UserWriteCtxSubmit 流程

创建一系列局部变量 通过`LuGet`使用req的LU id返回user(UserLu_t *)，创建iter(LrangeEnum_t)，创建attr(CmdAttr_t)...

如果req的id小于MAX_FTL_REQ(正常io)： 将传入的ctx(hostCtx)转换作为(FtlIO_Ctx_t*)解析 为一系列的局部变量赋值(meta、dataBuffer、status、Paa)

根据host_ctx中的status和PgaStatus_t的大小(Aspb等于两个spb：在物理上两个spb共享一个channel)获取页

将`PageAllocatorGet`返回给AllocCnt(分配页计数)

- PageAllocatorGet

首先通过`PgaAssignPaa`获取计数值(get_cnt)

根据get_cnt和skip_page进入， 根据page status的(cnt+skip)不等于0和slot是否等于当前slot进行判断，若真，重新计算一下index和SlotId

然后给page status的cnt、skip、slot参数赋值，index增加。

- PgaAssignPaa

创建aspb指针指向当前(`PgaGetCurActiveSpb`)Active的spb和其他一些变量

根据aspb指向的spb id和pda的索引通过`nal_make_pda`返回一个pda(physical d address)

循环：(从当前在页内的写指针小于其页最大数量的情况下)

```
将LPaa `TransPaa`到WPaaOff(Details在函数`TranslatePaa`中)
判断是否是DefectPage 若是则跳过该页

根据get_cnt进行一些条件判断

根据WPaaOff
获取Die id(`nal_pda_to_tgt`)
获取当前页的index
设置PageCnt
LPaa 加上一个DU_Page大小\4
aspb的write ptr和Occupy计数增加
get_cnt也增加
```

循环结束后进行一些值的改变，退出PgaAssignPaa。

从PageAllocatorGet出来后，根据aspb当前Slot的mode进行配置attr的slcmode、cahcemode参数

然后进行setup_meta(设置meta数据)

进行LbaRangeEnumInit(初始化 配置iter的参数)

进入循环：(循环MAX_ASPB_PER_PGA 两次)

数据类型

LogicUnit_t

```
typedef struct _LogicUnit_t
{
    U8 LuId;
    U8 LuFlags;
    U16 IoBlockFlags;
    U32 Capacity; // 容量
    U32 sn;
    BlkQueue_t BlkQueue;
    GcReq_t gcReq;
    SysLogDesc_t Sld;
    SldFlush_t SldFlush;

    LogicUnitMethod_t intf; // 方法函数
}LogicUnit_t;
```

UserLu_t

```
typedef struct _UserLu_t
{
    LogicUnit_t Lu;
    U32 Flags;
    PageAllocate_t PageAlloc[USER_MAX_PAGE_ALLOC]; // 包含PageAllocate方法函数
    SpbAppl_t SpbAppl[USER_MAX_PAGE_ALLOC]; // 包含VerifyFunc NotifyFunc
AllocFunc三种方法函数
    U32 u32ReadTraining;

    L2ppRa_t l2ppRa;

    CurWrCtx_t CurWrCtx;
    AdminCmdCtx_t AdminCtx;

    Lut_t lut;
    RlutLu_t rlutLu[USER_MAX_PAGE_ALLOC];

    ConvertReq_t creq;
    Wait_t *GcCvrtSync;
    AspbInfo_t AspbInfo[USER_MAX_PAGE_ALLOC][MAX_ASPB_PER_PGA];

    FC_t fc; // flow control
}UserLu_t;
```

_SrbMgrVirginInit

Write_GetNewCmd

Cache_GetCache

WriteResp_GetResp

ReadGetNewCmd

Resp_GetResp

RPMBGetNewCmd

Req_GetNew

FFlush_GetNew