

实验案例四：内核子系统—进程管理

实验案例四：内核子系统—进程管理

- 一、实验简介
- 二、实验内容及要求
 - 1.进程信息打印
- 三、实验原理
 - 1. 进程控制块
 - 2. 任务(线程)控制块
 - 3. 进程与线程调度
- 四、实验流程
 - 任务一：进程信息打印
 - 1. 流程
- 五、参考资料

一、实验简介

在进行 OpenHarmony 进程管理实验之前，需要理解进程并非所有操作系统内核通用的概念。OpenHarmony 当前支持的三种内核中，只有 **Linux** 和 **LiteOS-A** 提供进程机制；而 **LiteOS-M** 内核由于缺乏 MMU，无法实现不同进程间的地址空间隔离，仅提供任务（tasks）机制。

本实验基于 LiteOS-A 内核，旨在学习和理解其进程管理机制。实验过程中需深入阅读 LiteOS-A 源码，并在 OpenHarmony 内核子系统中进行编程，实现与进程管理相关的功能，从而加深对 LiteOS-A 内核设计与实现的理解。

二、实验内容及要求

本次实验需要依次完成下列实验要求，并提交最终在qemu中的运行结果截图。

1.进程信息打印

本实验基于实验案例三的系统调用实验。在此基础上，需要修改 LiteOS-A 内核中处理系统调用中断的函数实现，使得在调用新创建的系统调用时，能够返回当前系统中所有正在运行的进程信息，并打印至控制台。实验预期效果如下：

```
OHOS:/$ ./bin/print_taskinfo
User mode:: SYS_print_taskinfo
Kernel mode: SYS_print_taskinfo
```

PID	PPID	PGID	Mode	Status	Priority	PName
0	2	2	kernel	Pending	31	KIdle
1	-1	1	user	Pending	28	init
2	-1	2	kernel	Pending	0	KProcess
3	1	3	user	Pending	28	mksh
5	3	5	user	Pending	28	print_taskinfo

三、实验原理

1. 进程控制块

作为保存进程相关信息，进程控制块(PCB)是进程存在的唯一标志。在OpenHarmony的LiteOS-A内核之中，进程控制块结构体的定义保存在文件 `//kernel/liteos_a/kernel/base/include/los_process_pri.h` 之中的 `LosProcessCB`结构体内，部分内容如下：

```
typedef struct ProcessCB {
    CHAR                processName[OS_PCB_NAME_LEN]; /**< Process name */
    UINT32              processID;                    /**< Process ID */
    UINT16              processStatus;                /**< [15:4] Process Status; [3:0] The
number of threads currently
running in the process */
    UINT16              consoleID;                    /**< The console id of task belongs
*/
    UINT16              processMode;                  /**< Kernel Mode:0; User Mode:1; */
    struct ProcessCB    *parentProcess;               /**< Parent process */
    UINT32              exitCode;                     /**< Process exit status */
    LOS_DL_LIST         pendList;                     /**< Block list to which the process
belongs */
    LOS_DL_LIST         childrenList;                  /**< Children process list */
    LOS_DL_LIST         exitChildList;                 /**< Exit children process list */
    LOS_DL_LIST         siblingList;                   /**< Linkage in parent's children
list */
    ProcessGroup        *pgroup;                      /**< Process group to which a process
belongs */
    LOS_DL_LIST         subordinateGroupList;          /**< Linkage in group list */
    LosTaskCB          *threadGroup;                  /**< List of threads under this
process */
    volatile UINT32     threadNumber; /**< Number of threads alive under this process */
    UINT32              threadCount; /**< Total number of threads created under this
process */
    LOS_DL_LIST         waitList; /**< The process holds the waitLists to support
wait/waitpid */
    ...
} LosProcessCB;
```

其中几个结构体成员作用如下：

- processID: 进程的唯一标识符，通过此ID可指定与找到唯一的一个进程。
- processStatus: 进程的当前状态。
- parentProcess: 指向当前进程的父进程的指针。
- pendList: 保存了进程当前所属的阻塞列表，可通过此成员遍历此阻塞列表的所有成员。
- threadGroup: 指向属于当前进程的所有线程的首个线程控制块。
- threadSiblingList: 线程列表，保存着当前进程中所有线程控制块的地址，可通过此成员遍历进程的线程。

事实上在LiteOS-a操作系统中所有的进程控制块都被保存在一个声明在 `los_process_pri.h` 的 `g_processCBArray` 指针之中，在操作系统的C语言启动阶段之中会调用 `OsProcessInit()` 函数进行进程的初始化工作。在此之中，会为 `g_processCBArray` 申请一个能容纳 `g_processMaxNum(64)` 个进程控制块的空间。这就是整个LiteOS-a操作系统之中所能同时容纳的进程数量。

```
extern LosProcessCB *g_processCBArray;
extern UINT32 g_processMaxNum;
```

2. 任务(线程)控制块

类似于进程控制块，在LiteOS-a系统之中每个任务也同样需要一种控制块用于保存线程的相关信息，而这种结构体在LiteOS-a系统之中就定义在文件 `//kernel/liteos_a/kernel/base/include/los_sched_pri.h` 的 `LosTaskCB` 结构体之中，部分内容如下：

```
typedef struct TagTaskCB {
    VOID          *stackPointer;      /**< Task stack pointer */
    UINT16        taskStatus;         /**< Task status */

    UINT64        startTime;          /**< The start time of each phase of task */
    UINT64        waitTime;           /**< Task delay time, tick number */
    UINT64        irqStartTime;       /**< Interrupt start time */
    UINT32        irqUsedTime;        /**< Interrupt consumption time */
    INT32         timeslice;          /**< Task remaining time slice */
    SortLinkList  sortList;           /**< Task sortlink node */
    const SchedOps *ops;
    SchedPolicy   sp;

    UINT32        stackSize;          /**< Task stack size */
    UINTPTR       topOfStack;         /**< Task stack top */
    UINT32        taskID;             /**< Task ID */
    TSK_ENTRY_FUNC taskEntry;         /**< Task entrance function */
    VOID          *joinRetVal;        /**< pthread adaption */
    VOID          *taskMux;           /**< Task-held mutex */
    VOID          *taskEvent;         /**< Task-held event */
    UINTPTR       args[4];            /**< Parameter, of which the maximum number is 4 */
    CHAR          taskName[OS_TCB_NAME_LEN]; /**< Task name */
    LOS_DL_LIST   pendList;           /**< Task pend node */
    LOS_DL_LIST   threadList;         /**< thread list */
    ...
} LosTaskCB;
```

本节仅介绍与实验相关的部分结构体成员，如需深入了解，可参考源码进行进一步学习：

- `taskStatus`：保存线程当前状态。
- `ops`：保存着与任务调度策略相对应的不同函数，根据任务的调度策略不同，其指向不同的操作函数。
- `sp`：保存着当前任务所采用的调度策略相关信息，例如高优先级优先调度(HPF)所需使用到的当前任务优先级，以及最早截止时间优先(EDF)策略所使用的任务完成时间等信息。
- `taskID`：任务标识符，每一个ID都对应着唯一的一个任务控制块。
- `taskEntry`：指向任务所需要执行的函数。

- taskName: 任务名称。
- threadList: 保存着同属一个进程的所有姐妹任务的列表。

LiteOS-A 系统通过位于 `los_task_pri.h` 的全局变量 `g_taskCBArray` 保存所有任务控制块。在进程初始化阶段，`OsTaskInit()` 会为 `g_taskCBArray` 分配内存，可容纳最多 128 个任务控制块，因此系统在同一时间最多支持 128 个线程。

3. 进程与线程调度

OpenHarmony的LiteOS-a内核的调度需要使用调度队列，用于负责为CPU选择当前需要执行的任务，其中的结构体定义在 `los_sched_pri.h` 之中

```
typedef struct {
    SortLinkAttribute timeoutQueue; /* task timeout queue */
    HPFRunqueue        *hpfRunqueue;
    EDFRunqueue        *edfRunqueue;
    UINT64              responseTime; /* Response time for current CPU tick interrupts */
    UINT32              responseID;   /* The response ID of the current CPU tick interrupt */
    LosTaskCB           *idleTask;    /* idle task id */
    UINT32              taskLockCnt;  /* task lock flag */
    UINT32              schedFlag;    /* pending scheduler flag */
} SchedRunqueue;
```

在多核的环境下，操作系统每一个核都包含一个单独的调度队列。调度队列结构体之中都包含着IdleTask与两个子队列EDF与HDF调度队列。因此，在CPU中，每一个核都拥有着一个IdleTask，用于在没有其它任务空闲时执行。而其中使用到的两种任务队列hpfRunqueue和edfRunqueue,分别适用于几个不同的调度策略，分别代表着高优先级优先调度队列与最早截至时间优先调度队列。两者都定义在文件 `los_sched_pri.h` 之中，分别为HPFRunqueue与EDFRunqueue

```
#define OS_PRIORITY_QUEUE_NUM 32
typedef struct {
    LOS_DL_LIST priQueueList[OS_PRIORITY_QUEUE_NUM];
    UINT32      readyTasks[OS_PRIORITY_QUEUE_NUM];
    UINT32      queueBitmap;
} HPFQueue;

typedef struct {
    HPFQueue queueList[OS_PRIORITY_QUEUE_NUM];
    UINT32    queueBitmap;
} HPFRunqueue;

typedef struct {
    LOS_DL_LIST root;
    LOS_DL_LIST waitList;
    UINT64      period;
} EDFRunqueue;
```

正如上述代码所定义的那样，(1).对于EDF,最早截至时间优先调度队列，每次CPU调度时都从队列中的root链表中取第一个元素所指向的任务执行，不需要考虑任务的优先级。(2).对于HPF，也就是高优先级优先调度策略。其中使用到了两层的优先级队列，每一层的优先级共有32级，第一层为进程的优先级，第二层为线程优先级。CPU调度任务时，首先在第一层中依次寻找存在任务的高优先级队列，接着在该层队列中再依次寻找高优先级的子队列进行任务的调度。

```
#define LOS_SCHED_NORMAL    0U    // 默认调度策略，设置为时间片轮转
#define LOS_SCHED_FIFO     1U
#define LOS_SCHED_RR       2U
#define LOS_SCHED_IDLE     3U
#define LOS_SCHED_DEADLINE 6U
```

LiteOS-a内核之中，可选的调度策略总共包括有五种：(1).LOS_SCHED_IDLE策略仅用于Idle任务，当CPU空闲且无其他任务可运行时执行。(2).LOS_SCHED_DEADLINE策略为最早截止时间优先调度策略，该策略的任务将会放入到上文所提到的EDF队列之中。(3).LOS_SCHED_FIFO为先来先优先调度策略，LOS_SCHED_RR为时间片轮转调度策略，LOS_SCHED_NORMAL为系统任务的默认调度策略，相当于LOS_SCHED_RR。操作系统中使用LOS_SCHED_FIFO与LOS_SCHED_RR策略的任务将会根据情况放入到上文提到的HPF调度队列之中。具体来说，两种策略都是根据优先级将任务插入到priQueList末尾，但是先来先优先调度策略(LOS_SCHED_FIFO)在创建任务时为任务分配的时间片足够大，使得使用FIFO调度策略的任务执行过程中不会耗尽时间片。而采用了时间片轮转策略RR的任务，则每次插入调度就绪队列时，都只能根据当前队列中的任务数量，平均分配可用的时间片，轮流占用CPU资源。

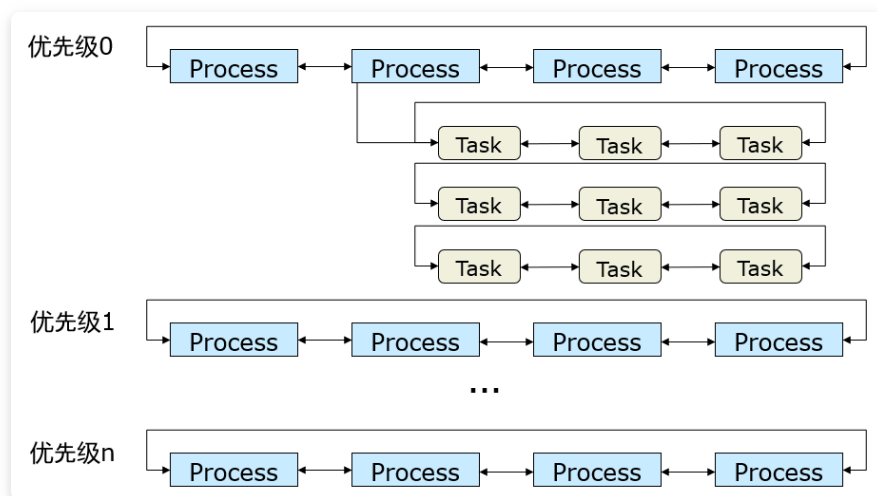


图1.LiteOs-a调度模型[1]

对于不同的策略下的任务调度具体操作，定义在los_priority.c的g_priorityOps与los_deadline.c的g_deadlineOps内，具体使用哪一种函数取决于任务控制表的ops成员,可根据需要自行阅读。

```
const STATIC SchedOps g_priorityOps = {
    .dequeue = HPFDequeue,
    .enqueue = HPFEnqueue,
    .waitTimeGet = HPFWaitTimeGet,
    .wait = HPFWait,
    .wake = HPFWake,
    .schedParamModify = HPFSchedParamModify,
    .schedParamGet = HPFSchedParamGet,
    .delay = HPFDelay,
    .yield = HPFYield,
    .start = HPFStartToRun,
    .exit = HPFExit,
```

```
.suspend = HPFSuspend,  
.resume = HPFResume,  
.deadlineGet = HPFTimeSliceGet,  
.timeSliceUpdate = HPFTimeSliceUpdate,  
.schedParamCompare = HPFParamCompare,  
.priorityInheritance = HPFPriorityInheritance,  
.priorityRestore = HPFPriorityRestore,  
};
```

四、实验流程

任务一：进程信息打印

1. 流程

本节实验在上一节系统调用实验的基础上进行，需要为已实现的系统调用函数增加功能，使其能够读取当前操作系统环境下所有进程和线程的信息，并打印输出。具体步骤如下：

1. 阅读 LiteOS-A 内核中与进程和线程相关的源码，理解 OpenHarmony LiteOS-A 内核如何管理进程与线程，以及任务调度的实现机制。
2. 修改上一节新增的系统调用处理函数，使其在打印 "kernel mode:: SYS_print_taskinfo" 字段后，能够收集系统中所有存在的进程及其线程信息，并按实验要求格式输出。
3. 编译并进入 OHOS 系统，运行新增组件程序，验证其是否能够正确打印操作系统中所有进程和线程信息。

五、参考资料

[1]. 进程管理模块:<https://docs.openharmony.cn/pages/v4.1/zh-cn/device-dev/kernel/kernel-small-basic-process-process.md>