项目二：向 LiteOS 中添加一个简单的基于线程运行时间的短作业优先调度策略

22920212204396　　黄子安

一、实验目的

1、理解 Liteos-a 中的任务调度，增加一个短作业优先策略（SJF, Shortest Job First）

二、实验环境

1.物理机：windows 操作系统

2.VMware 虚拟机：ubuntu 18.04.6

3.开发板：imx6ull Mini

三、实验内容

0.前置说明

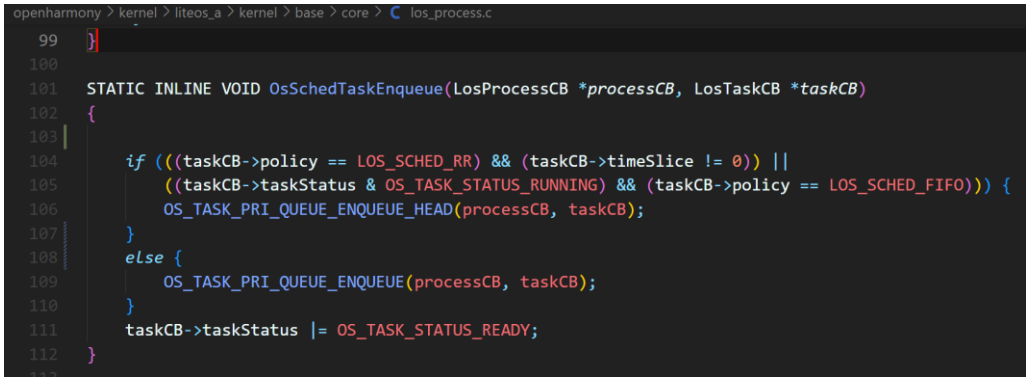
本次实验会向鸿蒙中增加短作业优先调度策略，但是依旧会受到时钟中断的影响（具体在后续过程说明），并且作用的范围是我们在应用程序中通过 pthread 创建的三个线程，这三个线程我们会提供一个预测运行时间，之后预测时间最短的线程先完成执行，再依次执行预测时间稍微长一些的线程，因此是会出现饥饿现象

1.增加调度算法

实验的第一步是先往 TCB 中增加一个 predict_time 字段，用于记录预测执行时间，TCB 被定义在 openharmony\kernel\liteos_a\kernel\base\include\los_task_pri.h 中，这里可以看到有一个字段为 policy 存储调度的策略

```
296 typedef struct {
297     VOID            *stackPointer;      /**< Task stack pointer */
298     UINT16          taskStatus;         /**< Task status */
299     UINT16          priority;           /**< Task priority */
300     UINT16          policy;
301     UINT16          timeSlice;          /**< Remaining time slice */
302     UINT16          predict_time;
303     UINT32          stackSize;          /**< Task stack size */
304     UINTPTR         topOfStack;         /**< Task stack top */
305     UINT32          taskID;             /**< Task ID */
306     TSK_ENTRY_FUNC  taskEntry;          /**< Task entrance function */
307     VOID            *joinRetval;        /**< pthread adaption */
308     VOID            *taskSem;           /**< Task-held semaphore */
309     VOID            *taskMux;           /**< Task-held mutex */
310     VOID            *taskEvent;         /**< Task-held event */
311     UINTPTR         args[4];            /**< Parameter, of which the maximum
```

之后我们要修改的就是 TCB 插入到就绪队列的方式，原先在内核中提供了两种方式，一种是时间片轮转（RR），另一种是先来先到（FIFO），该部分代码位于 openharmony/kernel/liteos_a/kernel/base/core/los_process.c 中

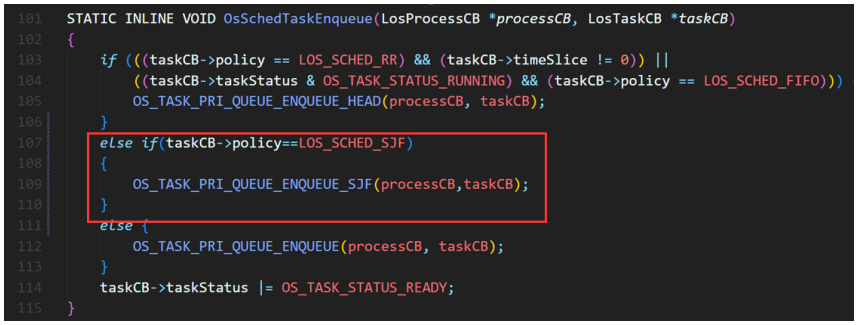


这段代码的解释如下：

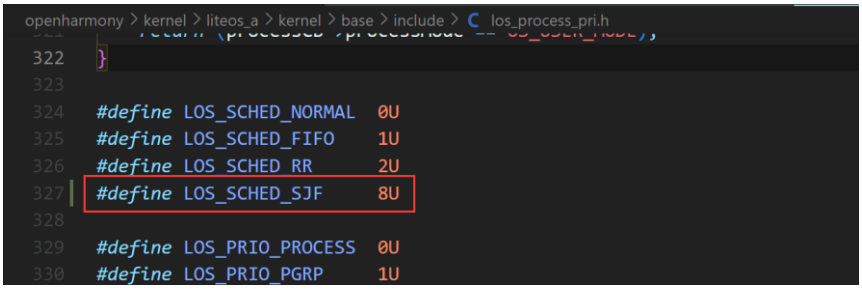
- 如果是时间片轮转策略并且时间片没用完（被抢占或阻塞导致）就放在对应优先级队列的头部，如果时间片用完了就放在最后；
- 如果是 FIFO 策略并且是运行态，即刚从 CPU 上下来，那应该让它回到队列的头部继续运行，否则放到队尾

这段代码会在两个情况执行，一个情况是新的 TCB 被创建的时候会将其**放入**就绪队列；另一个情况是时钟中断到达，就会先把运行的 TCB **放回**就绪队列，之后再重新选出最高优先级的进行调度

因此，对于我们要实现的 SJF 代码，效果是在创建的时候根据它的预测时间将其放到合适的位置；如果是运行完的 TCB 也会再次根据它的预测时间来重新插入回就绪队列。如果不被发生抢占的情况，当前预测时间最短的会回到头部，因此会重新又一次上 CPU，直到其被运行完。



之后去内核中补充进这个调度策略的宏定义，对于的宏定义位置如下图所示



接下去实现这个调度策略的具体算法，先在 los_task_pri.h 文件中添加对应的宏

```
openharmony > kernel > liteos_a > kernel > base > include > C los_task_pri.h
454     OsPriQueueEnqueueHead((processCB)->threadPriQueueList, &((processCB)->threadScheduleMap), \
455                           &((taskCB)->pendList), (taskCB)->priority)
456
457 #define OS_TASK_PRI_QUEUE_ENQUEUE_SJF(processCB, taskCB) \
458     OsPriQueueEnqueueSJF((processCB)->threadPriQueueList, &((processCB)->threadScheduleMap), \
459                          &((taskCB)->pendList), (taskCB)->priority)
460
```

在文件 openharmony/kernel/liteos_a/kernel/base/include/los_priqueue_pri.h 中添加该函数的声明

```
156 extern VOID OsPriQueueEnqueueSJF(
157     LOS_DL_LIST *priQueueList, UINT32 *bitMap, LOS_DL_LIST *priqueueItem, UINT32 priority);
```

之后再去文件 los_priqueue.c 中实现具体的算法，该函数总共是四个参数，第四个参数是 UINT32 priority。该算法的思路就是比较优先队列中当前优先级下各个 TCB 中的预测时间和待插入 TCB 的预测时间（代码在附录中给出文字形式）

```
openharmony > kernel > liteos_a > kernel > base > sched > sched_sq > C los_priqueue.c
93
94  VOID OsPriQueueEnqueueSJF(LOS_DL_LIST *priQueueList, UINT32 *bitMap, LOS_DL_LIST *priqueueItem, UINT3
95  {
96      LOS_ASSERT(priqueueItem->pstNext == NULL);
97      LosTaskCB *newtask=LOS_DL_LIST_ENTRY(priqueueItem, LosTaskCB, pendList);
98      if (LOS_ListEmpty(&priQueueList[priority])) {
99          *bitMap |= PRIQUEUE_PRIOR0_BIT >> priority;
100         LOS_ListTailInsert(&priQueueList[priority], priqueueItem);
101         return;
102     }
103     LOS_DL_LIST *item;
104     LosTaskCB *taskitem;
105     LOS_DL_LIST_FOR_EACH(item,&priQueueList[priority]){
106         taskitem=LOS_DL_LIST_ENTRY(item, LosTaskCB, pendList);
107         if(newtask->predict_time<taskitem->predict_time){
108             LOS_ListAdd(item->pstPrev,priqueueItem);
109             break;
110         }
111     }
112     if(!(newtask->predict_time<taskitem->predict_time)){
113         LOS_ListTailInsert(&priQueueList[priority], priqueueItem);
114     }
115 }
```

- 如果该优先级下这个循环双向链表是空的说明这个优先级下暂时还没有 TCB，直接插入即可
- 如果找到一个 TCB（taskitem）的预测时间大于当前待插入 TCB（newtask）的预测时间，就需要将待插入 TCB 放在这个 TCB 的前面，使用内核中的 LOS_ListAdd 即可实现，该函数定义如下

```
LITE_OS_SEC_ALW_INLINE STATIC_INLINE VOID LOS_ListAdd(LOS_DL_LIST *list, LOS_DL_LIST *node)
{
    node->pstNext = list->pstNext;
    node->pstPrev = list;
    list->pstNext->pstPrev = node;
    list->pstNext = node;
}
```

通过分析这个函数可以知道会插入到 list 节点的下一个位置，因此我们在传入参数的时候需要取 item 的前一个结点

- 如果当前循环链表中的预测时间都小于待插入的 TCB，则将这个 TCB 放在最后即可

经过上述修改内核便实现了支持 SJF 调度策略

2.设置测试线程

现在还有两个问题没有解决：

- 怎么设置一个任务的调度策略为 SJF
- 怎么设置一个任务的预测时间

这里的话有三种思路，但是实现起来都比较麻烦：

- 修改 pthread_create 函数，但是 libc 库是被静态编译成静态文件，然后在 clang 编译时和应用程序进行链接，想要重新编译这个库极其麻烦，具体可以参考黄忠同学的博客
- 内核中的 POSIX 接口提供了另一个 pthread 库，但是我们很难去直接使用这部分代码与应用程序编译链接，使用系统调用执行这部分代码经过证明会出现不少问题
- 绕开 pthread 库，通过系统调用直接调用 LOS_TaskCreateOnly 和后续的 LOS_SetTaskScheduler，这个方法相对比较方便，但是增加系统调用本身也会更改不少文件

本实验的目的是**了解就绪队列的插入机制**，从而实现 SJF 调度，而无论任何形式的 pthread_create 都是直接或间接调用了内核的 LOS_TaskCreateOnly 和 LOS_SetTaskScheduler，因此如果为了实验目的我们可以直接采用写死在代码里即可，而不陷入到修改 pthread 库的泥淖中

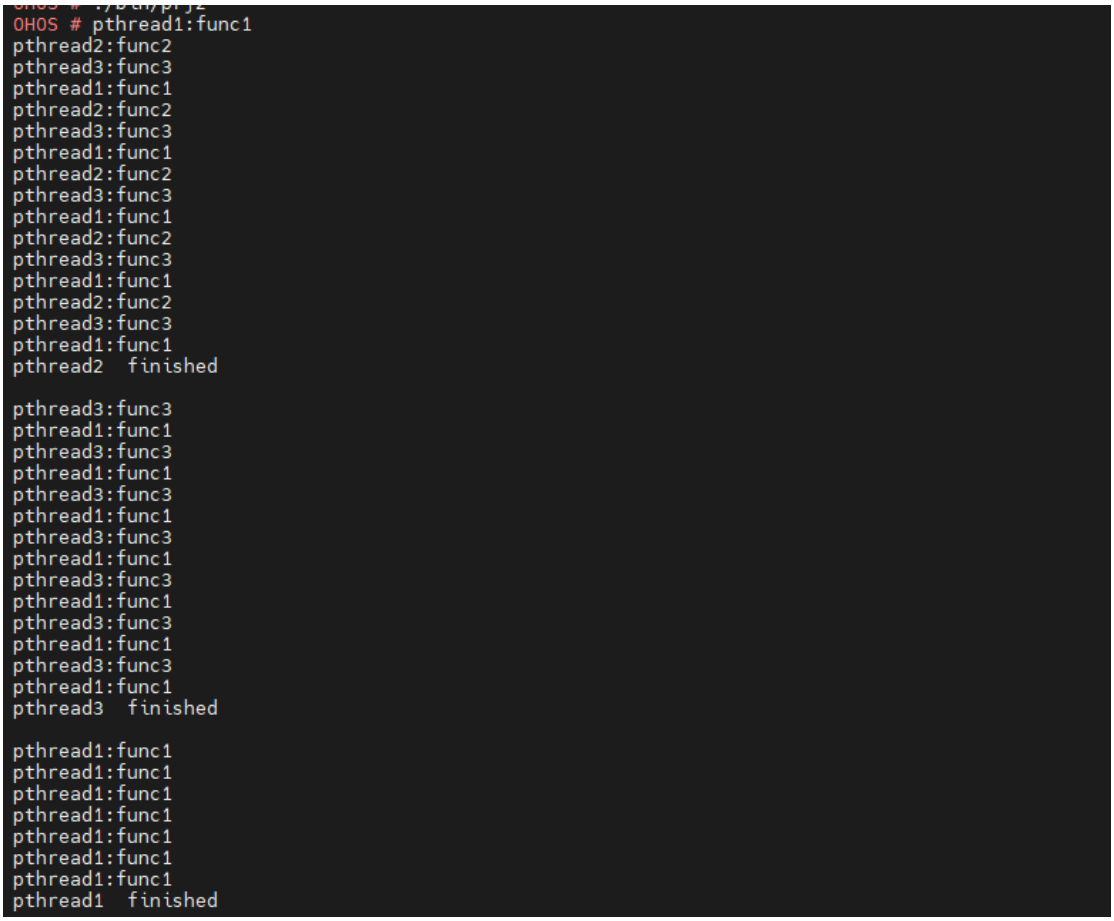
具体的实验方案就是经过实验证明可以发现我们如果在**开发板通电后不进行其他操作**的情况下直接在用户态下创建三个线程，则这三个线程的 TaskId 是**固定不变**的，每次分配到的 id 都是一样的，因此我们直接可以在内核判断这三个线程之后进行相关的赋值操作

我们先编写一个用于测试的应用程序 prj2（直接放在~目录里即可），具体使用的代码在附录中给出，这里我们**没有使用 sleep 函数**，是因为 sleep 函数会将线程放到阻塞队列，它重新回到就绪队列的时候不会调用我们之前的 OsPriQueueEnqueueSJF，因此我们直接使用 while 循环空转的方式来进行线程的等待

直接运行下这个程序，获取这三个线程的 TaskId，使用以下命令完成内核的编译和根文件目录的制作

```
//编译内核
cd    /home/book/openharmony/kernel/liteos_a
make clean
make -j 16
make rootfs
//交叉编译 prj2
cd ~
clang -target arm-liteos --sysroot=/home/book/openharmony/prebuilts/lite/sysroot/ -o prj2 prj2.c
cp    prj2    /home/book/openharmony/kernel/liteos_a/out/imx6ull/rootfs/bin
cd    /home/book/openharmony/kernel/liteos_a/out/imx6ull/
mkfs.jffs2    -s 0x10000 -e 0x10000 -d rootfs -o rootfs.jffs2
```

之后将生成的 liteos.bin 和 rootfs.jffs2 下载到烧写工具的 files 目录中，运行开发板后输入./bin/prj2 运行应用程序，可以看到这三个线程现在是时间片轮状的方式运行（线程 2 输出 5 次，线程 3 输出 12 次，线程 1 输出 20 次）



使用命令 task 查看这三个线程的 id（可以直接在程序 prj2 运行的时候使用输入 task 命令），可以看到三个线程的 id 是 12、13、14

```
task

  PID  PPID PGID       UID  Status VirtualMem ShareMem PhysicalMem CPUUSE10s  PName
    1    -1    1         0   Ready   0x1af000  0x2c000     0x1d29d       0.0  init
    2    -1    2         0    Pend   0x33d255        0    0x33d255       0.0  KProcess
    3     1    1         0 Running   0x1bc000  0x31000     0x28a9d       0.0  shell
    4     3    4         0   Ready   0x218000  0x2e000     0x1e29d      99.9  prj2

  TID  PID      Status StackSize WaterLine    MEMUSE  TaskName
    8    1       Ready    0x3000     0xa6c    0x5a70  init
    0    2        Pend    0x1000     0x1bc         0  ResourcesTask
    2    2        Pend    0x4000     0x204         0  Swt_Task
    3    2        Pend    0x4000     0x1bc         0  system_wq
    5    2        Pend    0x4000     0x1c4         0  jffs2_gc_thread
    6    2        Pend    0x6000     0x238         0  HdfTouchEventHandler
    7    2        Pend    0x4000     0x2ac         0  SendToSer
    4    3        Pend    0x3000     0x9e0    0x7188  shell
    9    3     Running    0x3000     0x714   0x32454  ShellTask
   10    3        Pend    0x3000     0x4e8     0x1f0  ShellEntry
   11    4       Ready    0x3000     0x9fc    0xa1a8  prj2
   12    4        Exit    0x3000     0x3ac      0x90  thread1
   13    4        Exit    0x3000     0x3ac         0  thread2
   14    4        Exit    0x3000     0x3ac         0  thread3
OHOS #
```

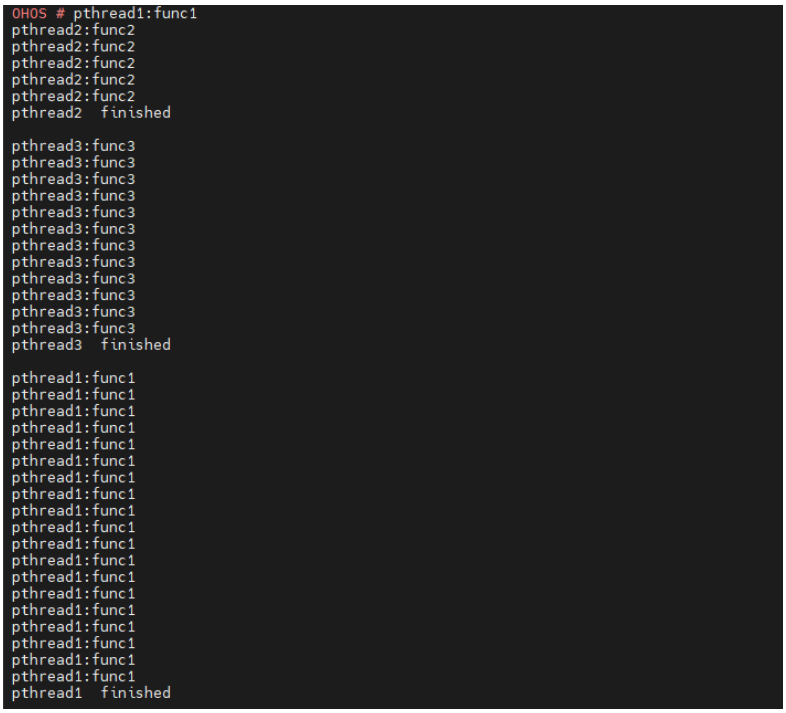
之后修改下代码在入队的时候进行特判，再重新编译内核与应用程序，下载到开发板上运行

```
openharmony > kernel > liteos_a > kernel > base > core > C los_process.c
 99  }
100
101 STATIC INLINE VOID OsSchedTaskEnqueue(LosProcessCB *processCB, LosTaskCB *taskCB)
102 {
103     if(taskCB->taskID==12) taskCB->predict_time=9;
104     if(taskCB->taskID==13) taskCB->predict_time=1;
105     if(taskCB->taskID==14) taskCB->predict_time=5;
106     if(taskCB->taskID==12||taskCB->taskID==13||taskCB->taskID==14)
107     {
108         OS_TASK_PRI_QUEUE_ENQUEUE_SJF(processCB, taskCB);
109     }
110     else
111     {
112         if (((taskCB->policy == LOS_SCHED_RR) && (taskCB->timeSlice != 0)) ||
113             ((taskCB->taskStatus & OS_TASK_STATUS_RUNNING) && (taskCB->policy == LOS_SCHED_FIFO))) {
114             OS_TASK_PRI_QUEUE_ENQUEUE_HEAD(processCB, taskCB);
115         }
116         else {
117             OS_TASK_PRI_QUEUE_ENQUEUE(processCB, taskCB);
118         }
119     }
120     taskCB->taskStatus |= OS_TASK_STATUS_READY;
121 }
```

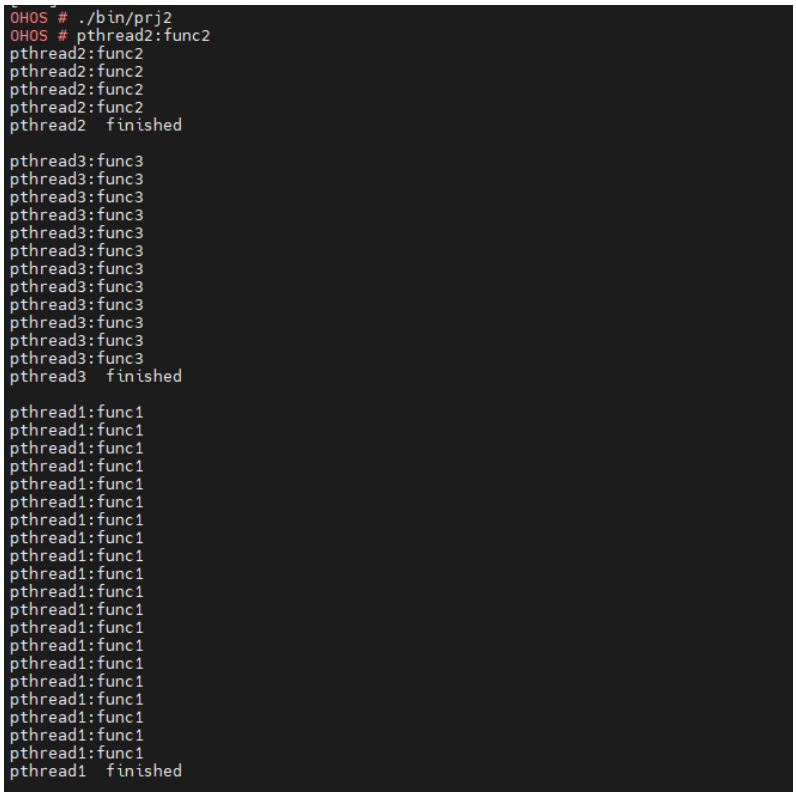
四、实验结果

运行对应的应用程序，可以发现可能会出现两种情形

第一种情形是先运行一下了 1 号线程，然后再依次运行 2 号线程、3 号线程和 1 号线程



第二种情形是依次运行 2 号线程、3 号线程和 1 号线程，这个情况概率比较小，第一次做实验运气好出现了，但是后来复现的时候尝试了 10 几次才出现



五、实验分析

上述两种情形都是正确的，第二种情形符合预期，第一种情形是因为在 1 号线程进入就绪队列后**主线程的时间片恰好到了**，这个时候 2 号、3 号线程还没创建，因此 1 号线程直接上 CPU 运行了，后续 2 号线程创建出来就会抢占掉 1 号线程，因此是没有问题的,成功实现短作业优先调度策略。

六、实验总结

本次实验实现 SJF 算法本身不算困难，只要对内核的数据结构有一定了解即可实现，但是在纠结修改 pthread_create 的时候花费了大量时间和精力，不过这个探索过程学到了不少 libc 库、posix 接口和交叉编译的知识，也算是非常有收获。

另外通过设计实验的方式确实可以对阅读代码起到非常大的帮助，通过动手和阅读结合的方式能更好的理解代码。

七、参考文献

[1] https://blog.csdn.net/qq_46120612/article/details/134358872

[2] https://blog.csdn.net/pingxiaozhao/article/details/122224969

[3] https://blog.51cto.com/u_14940441/2729241

八、附录

```
//prj2.c
#include <pthread.h>
#include <stdio.h>
#include <sched.h>
void s()
{
    int T=1e7;
    while(T--);
}
void * func1()
{
    int t=20;
    while(t--){
        printf("pthread1:%s\n",__func__);
        s();
    };
    printf("pthread1  finished\n\n");
}
void * func2()
{
    int t=5;
    while(t--){
        printf("pthread2:%s\n",__func__);
        s();
    }
    printf("pthread2  finished\n\n");
}
void * func3()
{
    int t=12;
    while(t--){
        printf("pthread3:%s\n",__func__);
        s();
    }
    printf("pthread3  finished\n\n");
}
int main()
{
    pthread_attr_t attr1,attr2,attr3;
    pthread_t ppid1,ppid2,ppid3;
```

```
    pthread_attr_init(&attr1);
    pthread_attr_init(&attr2);
    pthread_attr_init(&attr3);
    pthread_create(&ppid1,NULL,func1,NULL);
    pthread_create(&ppid2,NULL,func2,NULL);
    pthread_create(&ppid3,NULL,func3,NULL);

    while(1);//避免主线程退出
    return 0;
}
```

```
//openharmony/kernel/liteos_a/kernel/base/sched/sched_sq/los_priqueue.c
VOID OsPriQueueEnqueueSJF(LOS_DL_LIST *priQueueList, UINT32 *bitMap,
LOS_DL_LIST *priqueueItem, UINT32 priority)
{
    LOS_ASSERT(priqueueItem->pstNext == NULL);
    LosTaskCB *newtask=LOS_DL_LIST_ENTRY(priqueueItem, LosTaskCB,
pendList);
    if (LOS_ListEmpty(&priQueueList[priority])) {
        *bitMap |= PRIQUEUE_PRIOR0_BIT >> priority;
        LOS_ListTailInsert(&priQueueList[priority], priqueueItem);
        return;
    }

    LOS_DL_LIST *item;
    LosTaskCB *taskitem;
    LOS_DL_LIST_FOR_EACH(item,&priQueueList[priority])
    {
        taskitem=LOS_DL_LIST_ENTRY(item, LosTaskCB, pendList);
        if(newtask->predict_time<taskitem->predict_time)
        {
            LOS_ListAdd(item->pstPrev,priqueueItem);
            break;
        }
    }
    if(!(newtask->predict_time<taskitem->predict_time))
    {
        LOS_ListTailInsert(&priQueueList[priority], priqueueItem);
    }
}
```