嵌入式系统概论实验

EDF 算法的实现及其改进

石若非

141250108

1. 实验环境

RTOS：UCOSII VC 移植版

IDE 与运行环境：Visual Studio 2015（V120）

2. 基础的思路、方法与改进

1. 利用 TCB 的 OSTCBExtPtr 指针进行 TCB 模块的扩展，以保护原有结构防止出错
2. 在 os_core.c 文件中的 OS_InitTaskIdle 中加入自己提前定义的 idle 进程的扩展部分的数据，防止空指针错误
3. 在 ucos_ii.h 进行新增数据结构的声明，并将其中的 timer 进程和 state 进程的 enable 值设为 0，防止在系统运行中出现不必要的进程而报错
4. 在 cpu_cfg.h 中将每秒的时间片数设置为 1，方便执行与测试
5. 通过替换 OS_SchedNew()函数实现新的调度，而不改动多余的部分防止不可预料的错误发生
6. 堆操作在每个时间片结束后执行，防止遗漏（改进部分）

3. 重要数据结构：存放于 ucos_ii.h 文件中

```
typedef struct edf_tcb_ext {
    INT32U deadline;
    INT32U start;
    INT32U c_ticks;
    INT32U p_ticks;
    INT32U remain_ticks;
    int is_task_complete;
}edf_tcb_ext;
```

用于拓展原始 TCB 模块的结构，进程创建时作为 OSTCBExtPtr 与原进程块结合，既不破坏原有结构，又可以进行灵活的地扩展，这里的扩展内容会在之后的算法介绍部分进行解释

```
typedef struct heap_element {
    INT32U value;
    INT8U task_priority;
    INT32U another_value;
    INT16U id;
}heap_element;

typedef struct min_heap {
    heap_element* data;
    int len;
    int max_size;
}min_heap;

min_heap* createHeap(INT32U max_size);
void clearHeap(min_heap* heap);
int isHeapEmpty(min_heap* heap);
void insertHeapElement(min_heap* heap, heap_element element);
heap_element deleteMinElement(min_heap* heap);

min_heap* ready_task_heap;
min_heap* wait_task_heap;
```

这些是与最小堆操作相关的数据结构，每个堆中的元素包括 value、another_value（value 是最小堆中用作比较的值，这两个值的意义在就绪堆和等待堆中有一定区别，之后会对此进行解释，这里是为了更好地复用而如此命名）、进程 id、进程优先级 task_priority（这里的优先级只是一个用于映射到 OSTCBPrioTbl 的固定值，而非 EDF 调度中真正需要考虑的优先级）。堆的操作包括创建、插入一个元素、获得并删除堆顶元素、判断堆是否为空等函数，这些函数在 my_min_heap 中实现；最后进行了就绪任务最小堆和等待任务最小堆的声明

4. EDF 改进算法：就绪任务与等待任务同时进行优先队列处理与查找

该算法的优化思路详见 moodle 的声明或提交文件中的《EDF 算法改进》

具体实现如下：

- 系统开始运行时先将所有的非 idle 任务加入等待任务堆：

```
OS_TCB* p_current = OSTCBList;
```

```
    while (p_current->OSTCBPrio < OS_TASK_IDLE_PRIO) {
        heap_element element;
        element.task_priority = p_current->OSTCBPrio;
        element.value = ((edf_tcb_ext*)p_current->OSTCBExtPtr)->start;
        element.another_value = ((edf_tcb_ext*)p_current->OSTCBExtPtr)->deadline;
        element.id = p_current->OSTCBId;
        insertHeapElement(wait_task_heap, element);
        p_current = p_current->OSTCBNext;
    }
```

这里的操作为遍历 TCB 链表，为所有非 idle 任务创建堆元素，包括 id、task_priority、value（等待堆中的 value 代表该任务下一次就绪的时间）、another_value（等待堆中代表任务的下一个 ddl），等待堆创建完毕后，最早就绪的任务对应的元素会自然出现在堆顶

每个时间片结束时执行的操作为：

```
int is_complete = 0;
        ((edf_tcb_ext*)OSTCBCur->OSTCBExtPtr)->remain_ticks--;
        if (((edf_tcb_ext*)OSTCBCur->OSTCBExtPtr)->remain_ticks == 0) {
            deleteMinElement(ready_task_heap);
            edf_tcb_ext* p_ext = ((edf_tcb_ext*)OSTCBCur->OSTCBExtPtr);
            p_ext->remain_ticks = p_ext->c_ticks;
            p_ext->start = p_ext->start + p_ext->p_ticks;
            p_ext->deadline = p_ext->deadline + p_ext->p_ticks;
            is_complete = 1;
            heap_element element;
            element.task_priority = OSTCBCur->OSTCBPrio;
            element.value = p_ext->start;
            element.another_value = p_ext->deadline;
            element.id = OSTCBCur->OSTCBId;
            insertHeapElement(wait_task_heap, element);
            is_complete = 1;
        }

        INT32U current_time = OSTimeGet();
        while (!isHeapEmpty(wait_task_heap)&&wait_task_heap->data->value ==
current_time) {
            heap_element wait_element = deleteMinElement(wait_task_heap);
            heap_element ready_element;
            ready_element.value = wait_element.another_value;
            ready_element.task_priority = wait_element.task_priority;
```

```
            ready_element.another_value = wait_element.value;
            ready_element.id = wait_element.id;
            insertHeapElement(ready_task_heap, ready_element);
        }

        if (is_complete> 0) {
            ((edf_tcb_ext*)OSTCBCur->OSTCBExtPtr)->is_task_complete = 1;
        }
```

这里的操作包括：

1. 当前进程的需执行剩余时间片减 1
2. 若该进程在这个时间片结束了其这个周期的工作，则将其从就绪堆中删除，修改该进程的下一次就绪时间和 ddl 并为其创建对应的堆元素，插入等待堆中，同时将这个结束了周期内任务的进程的扩展 is_task_complete 部分设为真
3. 检查等待堆的堆顶元素的 value（即下一次就绪时间）是否与当前系统时间一致，若一致，则将其从等待堆中获取并删除，将该元素的 another_value（即 ddl）作为创建的新的就绪堆元素的 value，并进行 id 和 task_priority 的赋值，然后插入就绪堆中
4. 执行 3 直到等待堆的堆顶元素的 value 不再与当前系统时间一致

- 基于 EDF 的下一个进程的查找的实现：

```
static INT8U OS_EDFSched(void) {
    if (isHeapEmpty(ready_task_heap)) {
        return OS_TASK_IDLE_PRIO;
    }
    else {
        return ready_task_heap->data->task_priority;
    }
}
```

很简单明了的算法：若就绪堆不为空，则取堆顶元素对应的进程，若为空，则执行 idle 进程

- 每次中断结束后需执行的内容（基本为时钟中断）：

```
if (((edf_tcb_ext*)OSTCBCur->OSTCBExtPtr)->is_task_complete==0) {
                    INT8U next_task_prio = OS_EDFSched();
                    INT16U next_task_id = get_next_task_id();
                    if (next_task_prio != OSPrioHighRdy) {
                        INT16U preempted_task_id = OSTCBCur->OSTCBId;
                        OSPrioHighRdy = next_task_prio;
                        OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
                        ((edf_tcb_ext*)OSTCBHighRdy->OSTCBExtPtr)->is_task_complete
= 0;
                        APP_TRACE("\n%d\tpreempt\t\t%d\t%d", OSTimeGet(),
(int)preempted_task_id, (int)(next_task_id));
                    }
                }
```

当发生抢占而非进程的周期内任务完成时，利用堆实现的截止时间最早的进程查找函数，若与当前进程不一致则发生抢占并输出（这里做的处理时，若该事件是一个进程的周期内任务完成驱动的，则跳过这里的操作，交给 OS_Sched()函数处理）

- 每次任务进入 delay 状态需执行的内容（即 OS_Sched()函数）：

```
INT16U complete_task_id = OSTCBCur->OSTCBId;
            INT8U next_task_prio = OS_EDFSched();
            OSPrioHighRdy = next_task_prio;
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            INT16U next_task_id = get_next_task_id();
            ((edf_tcb_ext*)OSTCBHighRdy->OSTCBExtPtr)->is_task_complete = 0;
            APP_TRACE("\n%d\tcompleted\t%d\t%d",     OSTimeGet(),     complete_task_id,
next_task_id);
```

该函数的执行条件比较严格，无法在中断中和调度被锁时执行（包括时钟中断），因此无法显示调用，这里我选择在进程中使用 OSTimeDly()函数间接调用，这样每次进程的周期内任务完成，都会调用该函数，选择下一个执行的进程，将下一个进程的完成状态设为假，并输出 complete 事件的内容

- 循环进程的内容：

```
static void task1(void *pdata) {
```

```
    while(1) {
        while (((edf_tcb_ext*)OSTCBCur->OSTCBExtPtr)->is_task_complete==0) {
            //donothing
        }
        OS_ENTER_CRITICAL();
        //APP_TRACE("\nthis is task1");
        OS_EXIT_CRITICAL();
        OSTimeDly(((edf_tcb_ext*)OSTCBCur->OSTCBExtPtr)->start - OSTimeGet());
    }
}
```

进程在 is_task_complete 为假时不停执行，若为真则跳出循环，并 delay 到下一个就绪时间的到来，这样的改动使得上文提到的各种依据进程状态来执行的逻辑得以实现

5. 测试用例

进程的创建在 app.c 的 main()函数中执行：

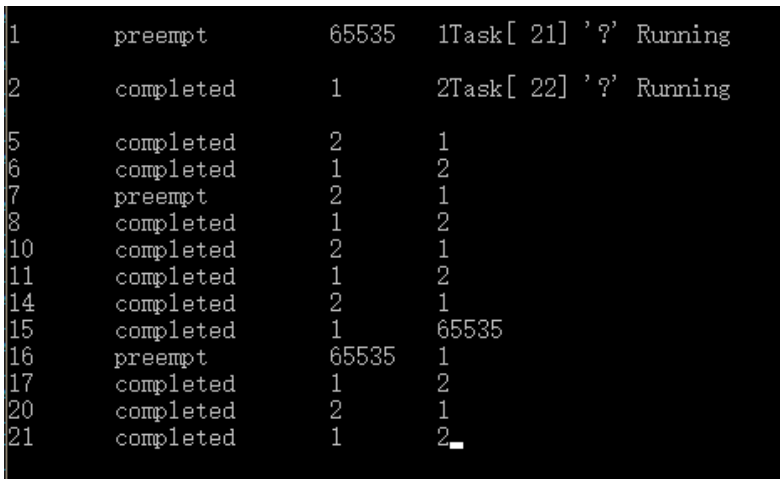
```
OSTaskCreateExt((void(*)(void *))task1,
        (void           *)0,
        (OS_STK          *)&TASK1STK[TASK_STK_SIZE - 1],
        (INT8U)TASK_1_PRIO,
        (INT16U)TASK_1_ID,
        (OS_STK         *)&TASK1STK[0],
        (INT32U)TASK_STK_SIZE,
        (void           *)&edf_exts[0],
            (INT16U)0);
```

测试用例选用了两个进程，分别为

```
//ddl,start,c,p,remain
edf_tcb_ext edf_exts[] =
{
  {4,1,1,3,1,0},
  {6,1,3,5,3,0},
};
```

分别为周期为 3，需执行 1 个时间片的和周期为 5，需执行 3 个时间片的进程，进程的 ID 分别为 1 和 2，每个进程内部的操作已在上文有过详细描述

执行结果如下：



经计算可得，发生的抢占、完成事件均与理论结果相符