|  |
| --- |
|  |
| EDF调度实验报告 |
|  |

|  |
| --- |
| 杜天蛟 141250031  2016-11-26 |

# 基本EDF调度实现

## 思路与方法

* 在TCB中添加关于EDF实现的信息

typedef struct edf\_data {

INT32U c\_value; // the task needs to consume c\_value ticks in p\_value ticks as a period, const

INT32U p\_value; // const

INT32U comp\_time; // the task remain to consume in a period

INT32U ddl; // the deadline of a task

INT32U start;

INT32U end;

}EDF\_DATA;

其中c\_value和p\_value是常量，任务一旦创建就不会改变，comp\_time是一个周期内还剩多少ticks没有执行，ddl是当前该任务的deadline，这两个值会根据时钟和调度而更新。Start和end是为了计算comp\_time和ddl方便添加的变量。

TCB的结构体中有个名为OSTCBExtPtr的void\*指针，可以指向用户自定义的TCB extension.因此创建任务时只需要把初始化过的edf\_data这个结构体传入就可以了。

* 创建自己的任务函数：

static void task1(void \*pdata) {

OSTimeDly(1);

while (1) {

while (((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->comp\_time > 0) {

//do nothing

}

}

}

仿照app.c中的AppTaskStart()的创建方式创建任务。定义任务栈，定义优先级和ID，并把自己定义好的edf\_data传入。

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\_datas[0],

(INT16U)0);

为了让任务同时进入系统，在每个任务开始时都加上了OSTimeDly(1)

* 编写一个新的任务调度函数OS\_SchedEDF，在每个时钟周期内遍历任务列表，找到ddl最近的任务，并把该任务的优先级赋值给OSPrioHighRdy。这样系统就可以选择优先级为OSPrioHighRdy的任务执行了。

static void OS\_SchedEDF(void) {

OS\_TCB\* p\_current;

OS\_TCB\* edf\_ptcb;

int temp\_earliest\_deadline = 1000000;

int temp\_deadline = 0;

int isAllDelay = 1;

p\_current = OSTCBList;

edf\_ptcb = OSTCBList;

OS\_ENTER\_CRITICAL();

while (p\_current->OSTCBPrio != OS\_TASK\_IDLE\_PRIO) {

//if the task is ready and not delay

if (p\_current->OSTCBPrio<60 && p\_current->OSTCBDly == 0 && ((EDF\_DATA\*)p\_current->OSTCBExtPtr)->comp\_time > 0) {

temp\_deadline = ((EDF\_DATA\*)p\_current->OSTCBExtPtr)->ddl;

if (temp\_deadline < temp\_earliest\_deadline) {

temp\_earliest\_deadline = temp\_deadline;

edf\_ptcb = p\_current;

}

isAllDelay = 0;

}

p\_current = p\_current->OSTCBNext;

}

OSPrioHighRdy = edf\_ptcb->OSTCBPrio;

if (isAllDelay == 1) {

OSPrioHighRdy = OS\_TASK\_IDLE\_PRIO;

}

OS\_EXIT\_CRITICAL();

}

* 在OSTimeTick中修改当前任务的comp\_time，更新该任务的ddl,start,end等值。

OS\_ENTER\_CRITICAL();

((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->comp\_time--;

if (((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->comp\_time == 0) {

((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->ddl = ((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->ddl + ((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->p\_value;

((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->comp\_time = ((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->c\_value;

((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->end = OSTimeGet();

OSTCBCur->OSTCBDly =

((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->p\_value - (((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->end - ((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->start);

((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->start = ((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->start + ((EDF\_DATA\*)OSTCBCur->OSTCBExtPtr)->p\_value;

printf("\n%d\tComplete\t%d\t%d", OSTimeGet(),OSTCBCur->OSTCBId,getEDFNextID());

}

else {

if (OSTCBCur->OSTCBId != getEDFNextID()) {

APP\_TRACE("\n%d\tPreempt\t\t%d\t%d", OSTimeGet(),OSTCBCur->OSTCBId,getEDFNextID());

}

}

OS\_EXIT\_CRITICAL();

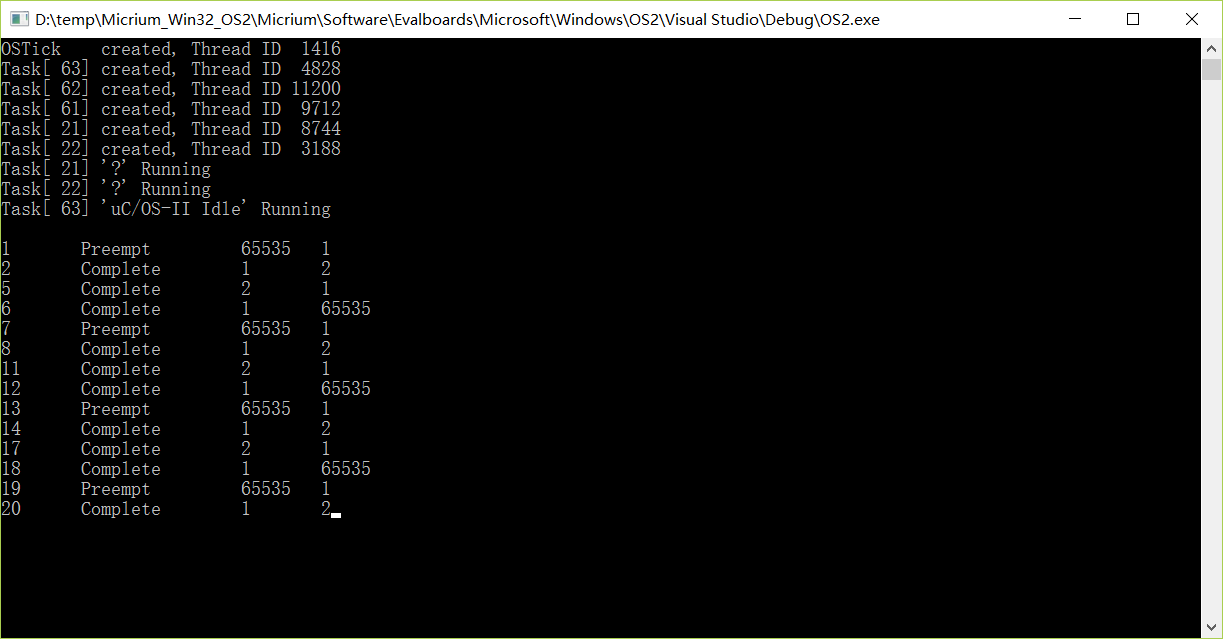
算法很简单，如果该任务的comp\_time为0，说明该任务已经完成，那就顺次后移ddl和start，重置comp\_time，否则不做改变。如果该任务的comp\_time不为0，则说明被抢占。为了将任务切换过程打印出来，又添加了getEDFNextID()的方法，其中算法和OS\_SchedEDF()中一样，找到下一个ddl最近的任务的Id并返回。

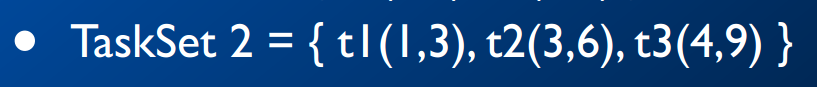
* 因为包的问题，有时候无法用UC/OS2定义好的APP\_TRACE的宏，所以我又在core.c上面#include<stdio.h>，使用printf来打印
* 为了方便查看输出结果，将os\_cfg.h中的OS\_TICKS\_PER\_SEC修改为1u

## 测试

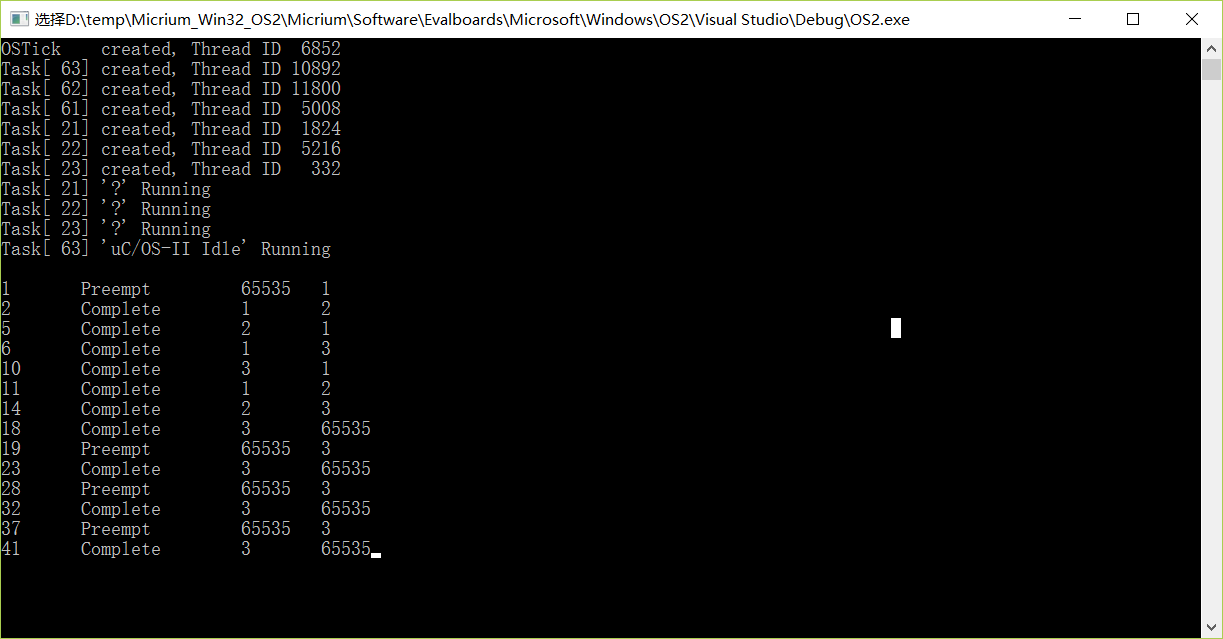


输出：





输出：



# EDF算法改进

## 算法描述

EDF 算法决定优先级的机制过于单一，仅仅依靠每个任务请求的绝对截止时间来，并不考虑任务是否重要，不能很好的反映实际要求。可能会出现在任务截止时间相同的情况下，一个软实时要求的任务得到调度，而另一个硬实时要求的任务得不到调度。

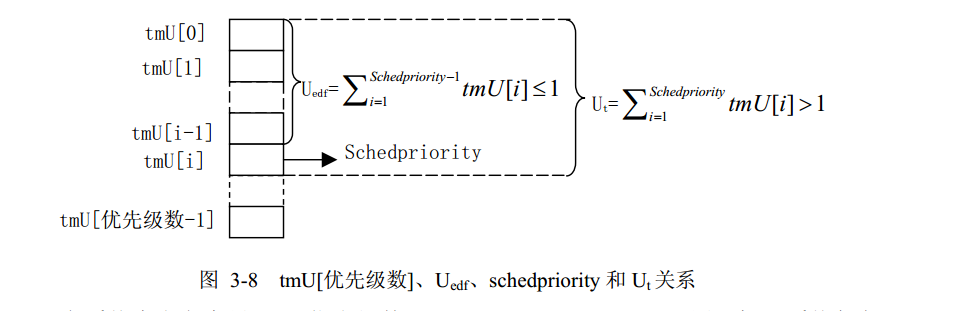
因此可以引入一个表示任务重要性的importance变量，用这个importance变量和任务的绝对截止时间来共同决定任务调度时的优先级。这样，当两个任务的ddl相同时，有硬实时要求的任务就会比有软实时要求的任务先得到调度。

改进后的EDF算法具体描述如下：

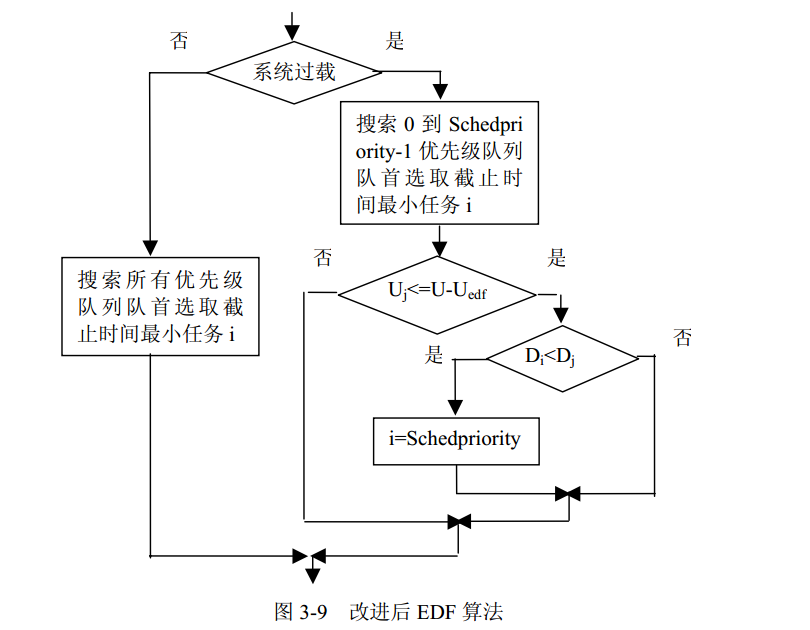
* 每个任务到来时需要提供周期、执行时间、优先级、Id和表示重要程度的level信息，其中level越小，意味着越重要。为每个level准备一个队列，用于存放本级别的就绪任务，任务在队列中按照绝对截止时间由小到大排列。对于截止时间相同的任务，level低的任务应在level高的任务前调度。

建立一个数组，数组的每个元素指向其下标所对应的任务level队列，可称之为level队列数组。就绪任务到来时先根据level找到对应的level队列，再根据截止时间加入该队列中。每个队列的队首任务就是本队列中绝对截止时间最小的任务。 在每个时钟周期内，系统搜索所有level队列的队首任务，选取截止时间最小的任务，并更新该队列。

* 如图所示，



在系统中定义变量 tmU[level]、 Uedf、 Schedpriority 和 Ut 用于标示系统任务中 EDF 算法可以调度的子任务集。tmU[level]用于记录每个level的所有任务CPU 利用率之和。在系统轻载情况下， Ut 是系统中所有level的就绪任务的 CPU利用率之和，即 Ut=∑i=1优先级数= tmU[i] ， n 为系统中所有任务的个数。系统过载情况下，Ut、 Uedf 和 Schedpriority 满足的关系如图 3-8 所示。其中 Uedf=∑i=1Schedpriority-1 tmU[i]≤1并且Uedf=∑i=1Schedpriority-1 tmU[i]>1。 Schedpriority 表示 EDF 算法可调度子任务集的level界限。level小于 Schedpriority 的任务可参与系统调度，level大于 Schedpriority 的任务不可参与系统调度。level等于 Schedpriority 的任务根据该任务的 CPU 使用率u 与 U-Uedf 的关系决定，当 u≤ U-Uedf 时可以参与系统调度；当 u > U-Uedf 时不参系统调度。当有新的任务到达或者需要从任务队列中删除任务时，系统需要重新计算 Uedf 和 Schedpriority。具体方法是：按level从低到高累加每个level的所有任务的 CPU 使用率，当累加值大于 1 时停止累加。 Schedpriority 等于此时的level。Uedf 等于累加值减去 tmU[Schedpriority]。计算 Uedf 和 Schedpriority 实际上是在对任务集进行可调度性判定。在综合考虑系统开销和判定效果后，本算法使用公式U=∑i=1nCi/Ti<=1(执行时间为Ci，周期为Ti)作为可调度性的判定依据。此外随着时间的流逝，一个任务的某次请求在可调度性判定之前没有得到调度，由于其剩余时间减小，此请求的紧迫程度增大，此刻的时限要求类似于周期为其剩余时间的周期任务（算法假设任务的截止时间等于周期）。所以在进行可调度性判定时使用此任务请求的剩余时间代替周期，即u’=Ci/(d-t)(t 为系统当前时间)。



* 对已经丢弃的任务，在整个实时系统恢复轻载状态时，其原有的deadline重新设置为当前时间t+ddl，重新进入调度。

## 改进算法实现

#### 关键数据结构

* typedef struct queue\_data {

INT32U prio;

INT32U deadline;

INT32U importance;

INT32U id;

}Queue\_Data;

Queue\_Data和每一个TCB对应，里面的prio,deadline,id和TCB中的对应属性始终保持一致。Importance属性表示该任务所属的等级

* typedef struct myqueue {

INT32U capacity;

INT32U size;

Queue\_Data \*elements;

}Queue;

Queue taskQueue[QUEUE\_LEVEL];

定义一个任务等级队列数组，每一个等级对应一个队列，elements指向该队列的队首。

void initQueue();

void enqueue(Queue\_Data\* data);

void update(INT32U level);

在系统初始化时调用initQueue()，为队列分配内存。

在创建任务时调用enqueue()，初始化该任务对应的Queue\_Data后，将其放在对应的队伍中去。（按ddl升序排）

在OSTimeTick()方法中，当该任务在该周期已经执行完毕，那么更新TCB属性后更新该任务所在队列的排序。

#### 核心算法

static void OS\_SchedEDFQueue() {

int i;

int earliest\_deadline = 1000000;

int earliest\_prio = 0;

int earliest\_level = 0;

OS\_ENTER\_CRITICAL();

for (i = 0; i < QUEUE\_LEVEL; i++) {

Queue\_Data\* first = taskQueue[i].elements;

if (first->deadline < earliest\_deadline) {

earliest\_deadline = first->deadline;

earliest\_prio = first->prio;

earliest\_level = i;

}

}

OS\_TCB\* p\_current = OSTCBList;

int isAllDelay = 1;

while (p\_current->OSTCBPrio != OS\_TASK\_IDLE\_PRIO) {

//if the task is ready and not delay

if (p\_current->OSTCBPrio<60 && p\_current->OSTCBDly == 0 && ((EDF\_DATA\*)p\_current->OSTCBExtPtr)->comp\_time > 0) {

isAllDelay = 0;

}

p\_current = p\_current->OSTCBNext;

}

OSPrioHighRdy = earliest\_prio;

level = earliest\_level;

if (isAllDelay == 1) {

OSPrioHighRdy = OS\_TASK\_IDLE\_PRIO;

}

OS\_EXIT\_CRITICAL();

}

即遍历每个队列的队首，找到ddl最小的一个任务执行。

## 测试

输入：



输出：

