# 各个调度算法的实现

## 抢占式优先级调度

### 算法简介

该算法根据任务的优先级进行调度，优先级高的任务优先执行。任务到达时，如果当前正在运行的任务优先级较低，则发生抢占。

### 算法实现

FreeRTOS自身支持抢占式优先级调度，仅需设置任务的优先级大小，系统会自动按照任务的优先级进行先后调度。

## 抢占式EDF调度

### 算法简介

在抢占式EDF调度中，任务根据其截止时间（Deadline）进行排序，优先执行Deadline最早的任务。如果新任务的截止时间更早，则当前任务会被抢占。

### FreeRTOS实现

#### 任务控制块定义

//任务控制块

typedef struct{

    uint32\_t ID;

    uint32\_t arrival\_time;  //到达时间

    uint32\_t execution\_time;    //执行时间

    uint32\_t period;    //周期

    uint32\_t deadline;  //截止时间

    uint32\_t weight;    //时间片权重

    TaskHandle\_t task\_handle;   //句柄

}Task\_Control\_Block;

#### 获取拥有最早deadline的任务

//获取拥有最早deadline的task

Task\_Control\_Block\* Get\_Earliest\_Deadline\_Task(void){

    int i;

    Task\_Control\_Block \*earliest\_task = NULL;

    uint32\_t min\_deadline = UINT32\_MAX;

    //搜索当前周期已经到达且为执行完毕的，拥有最近deadline的任务

    for(i=0;i<current\_task\_count;i++){

        if(tasks[i].arrival\_time <= current\_time && tasks[i].deadline < min\_deadline && tasks[i].execution\_time != 0){

            min\_deadline = tasks[i].deadline;

            earliest\_task = &tasks[i];

        }

    }

    return earliest\_task;

}

#### EDF调度器

//EDF调度器

void EDF\_Scheduler(void \*pvParameters)

{

        int i;

        char str[10];

//      LED0=!LED0;

    while (1) {

            Task\_Control\_Block\* task\_to\_execute = Get\_Earliest\_Deadline\_Task(); //获取当前时间片拥有最近deadline的任务

            show\_current\_time\_on\_lcd();

            //每个时间片开始前挂起所有任务

            for(i=0;i<current\_task\_count;i++){

                vTaskSuspend(tasks[i].task\_handle);

            }

//          printf("本轮调度任务：");

//          if(task\_to\_execute == NULL){

//

//          }

            LCD\_ShowString(150,700,110,16,16,(u8\*)"current time");

            LCD\_ShowxNum(250,700,current\_time/TIME\_SLICE\_INTERVAL,4,16,0x80);

            LCD\_ShowString(300,700,110,16,16,(u8\*)"current Task");

            LCD\_ShowxNum(400,700,task\_to\_execute->ID,8,16,0x80);

            if (task\_to\_execute != NULL) {

                //抢占式EDF

                for(i=0;i<current\_task\_count;i++){

                    if(&tasks[i] == task\_to\_execute){

                        vTaskResume(tasks[i].task\_handle);

                        break;

                    }

                }

                //可视化调度

                POINT\_COLOR = BLACK;

                for(i=0;i<current\_task\_count;i++){

                    LCD\_DrawRectangle(5+i\*150,110,115+i\*150,314);   //画一个矩形

                    sprintf(str, "task%d", i+1);

                    LCD\_ShowString(40+i\*150,350,60,16,16,(u8\*)str);

                    if(i+1 == task\_to\_execute->ID){

                        LCD\_Fill(6+i\*150,111,114+i\*150,313,RED); //填充区域

                    }

                    else{

                        LCD\_Fill(6+i\*150,111,114+i\*150,313,WHITE); //填充区域

                    }

                }

                task\_to\_execute->execution\_time -= TIME\_SLICE\_INTERVAL; //当前任务执行时间减少一个时间片

            }

            else{   //如果当前周期内所有任务执行完毕，挂起所有任务

                for(i=0;i<current\_task\_count;i++){

                    vTaskSuspend(tasks[i].task\_handle);

                }

                vTaskSuspend(EDF\_Scheduler\_Handle);

            }

            // 当前时间片+1

            current\_time+=TIME\_SLICE\_INTERVAL;

            vTaskDelay(TIME\_SLICE\_INTERVAL);//间隔TIME\_SLICE\_INTERVAL再执行

    }

}

## Round-Robin调度

### 算法简介

Round-Robin算法是时间片轮转算法，任务按顺序执行，每个任务分配固定的时间片，时间片用完后会被抢占并交给下一个任务。

### 算法实现

FreeRTOS自身支持时间片轮转算法，仅需设置任务的优先级大小，系统会自动按照任务的优先级进行先后调度。

## Weight-Round-Robin调度

### 算法简介

Weight-Round-Robin算法是对传统Round-Robin算法的扩展，为每个任务分配不同的时间片，任务的时间片与其权重成正比。

### FreeRTOS实现

#### 任务控制块定义

//任务控制块

typedef struct{

    uint32\_t ID;

    uint32\_t arrival\_time;  //到达时间

    uint32\_t execution\_time;    //执行时间

    uint32\_t period;    //周期

    uint32\_t deadline;  //截止时间

    uint32\_t weight;    //时间片权重

    TaskHandle\_t task\_handle;   //句柄

}Task\_Control\_Block;

#### W-R-R调度器

//Weight-Round-Robin调度器

void Weighted\_Round\_Robin\_Scheduler(void \*pvParameters){

    int i;

    int task\_index; //任务在数组中的序号

    int front\_task\_index;

    int task\_weight;

    while (1) {

        show\_current\_time\_on\_lcd();

        //每个时间片开始前挂起所有任务

        for(i=0;i<current\_task\_count;i++){

            vTaskSuspend(tasks[i].task\_handle);

        }

        //将已到达的任务入队,每个任务仅进行一次操作

        for(i=0;i<current\_task\_count;i++){

//          printf("a:%d at:%d\r\n",arrived\_tasks[i],tasks[i].arrival\_time);

            if(arrived\_tasks[i] == 0 && tasks[i].arrival\_time <= current\_time){

                task\_index = i;

                if(xQueueSendToBack(task\_Queue, &task\_index, portMAX\_DELAY) == pdPASS){

                    arrived\_tasks[i] = 1;

                    printf("task%d in. \r\n", tasks[i].ID);

//                  printf("task%u's weight: %u\r\n", tasks[i].ID, tasks[i].weight);

//                  if(i == 1){

//                      vTaskResume(tasks[i].task\_handle);

//                  }

                }else{

                    printf("Failed to enqueue task. \r\n");

                }

            }

        }

        if(uxQueueMessagesWaiting(task\_Queue) != 0){

            if(xQueueReceive(task\_Queue, &front\_task\_index, portMAX\_DELAY) == pdPASS){  //读取队首句柄

                if(front\_task\_index != -1){

                    printf("task%u gets %u time slices\r\n", tasks[front\_task\_index].ID, tasks[front\_task\_index].weight);

                    vTaskResume(tasks[front\_task\_index].task\_handle);

                    //判断是否带权

                    task\_weight = tasks[front\_task\_index].weight;

                    //让出时间片

                    vTaskDelay(task\_weight \* TIME\_SLICE\_INTERVAL);

                }else{

                    current\_time += TIME\_SLICE\_INTERVAL;

                    vTaskDelay(TIME\_SLICE\_INTERVAL);

                }

            }else{

                printf("Failed to dequeue task. \r\n");

            }

        }else{

            //队列中没任务则等待一个时间片

            current\_time += TIME\_SLICE\_INTERVAL;

            printf("当前时间片%d \r\n", current\_time/TIME\_SLICE\_INTERVAL);

            vTaskDelay(TIME\_SLICE\_INTERVAL);

        }

    }

}

## MLFQ算法

### MLFQ算法简述

MLFQ即Multi-Level Feedback Queue（多级反馈队列），是一种基于任务执行时间和优先级动态调整任务队列的调度算法，任务根据执行时间和等待时间在多个优先级队列间移动。

设有任务A（）和任务B（），MLFQ算法的基本逻辑如下：

1. 如果，则运行
2. 如果，则按照，所在队列的时间片进行Round-Robin调度
3. 当一个任务进入系统时，它将被放置到最高优先级的队列
4. 一旦任务用尽了它所在队列分配给它的时间片（无论它在运行期间放弃了多少次CPU），它的优先级都将被下调（即移动到优先级低一档的那个队列）
5. 经过一个指定时间S后，将所有任务移动至最高优先级的队列中。

### FreeRTOS实现

#### 任务控制块定义

//任务控制块

typedef struct{

    uint32\_t ID;

    uint32\_t arrival\_time;  //到达时间

    uint32\_t execution\_time;    //执行时间

    uint32\_t period;    //周期

    uint32\_t deadline;  //截止时间

    uint32\_t weight;    //时间片权重

    TaskHandle\_t task\_handle;   //句柄

}Task\_Control\_Block;

#### MLFQ调度器

void MLFQ\_Scheduler(void \*pvParameters){

    int i;

    int task\_index; //任务在数组中的序号

    int current\_task\_index;

    int temp;

    int boost\_flag = 1;

    while (1) {

        show\_current\_time\_on\_lcd();

        //每个时间片开始前挂起所有任务

        for(i=0;i<current\_task\_count;i++){

            vTaskSuspend(tasks[i].task\_handle);

        }

        //任务到达后进入最高优先级队列

        for(i=0;i<current\_task\_count;i++){

            if(arrived\_tasks[i] == 0 && tasks[i].arrival\_time <= current\_time){

                task\_index = i;

                if(xQueueSendToBack(Q3, &task\_index, portMAX\_DELAY) == pdPASS){

                    arrived\_tasks[i] = 1;

                    printf("T%d: task%d in. \r\n", current\_time/TIME\_SLICE\_INTERVAL, tasks[i].ID);

                }else{

                    printf("Failed to enqueue task. \r\n");

                }

            }

        }

        if((uxQueueMessagesWaiting(Q3) != 0 || uxQueueMessagesWaiting(Q2) != 0 || uxQueueMessagesWaiting(Q1) != 0) && current\_time != 0 && current\_time / (30 \* TIME\_SLICE\_INTERVAL) >= boost\_flag){

            printf("Priority boost!\r\n");

            boost\_flag++;

            //清空队列后，将所有任务置入最高优先级队列

            while(uxQueueMessagesWaiting(Q3) != 0 ){

                xQueueReceive(Q3, &temp, portMAX\_DELAY);

            }

            while(uxQueueMessagesWaiting(Q2) != 0 ){

                xQueueReceive(Q2, &temp, portMAX\_DELAY);

            }

            while(uxQueueMessagesWaiting(Q1) != 0 ){

                xQueueReceive(Q1, &temp, portMAX\_DELAY);

            }

            for(i=0;i<current\_task\_count;i++){

                if(arrived\_tasks[i] == 1 && tasks[i].arrival\_time <= current\_time && tasks[i].execution\_time != 0){

                    task\_index = i;

                    xQueueSendToBack(Q3, &task\_index, portMAX\_DELAY);

                }

            }

        }else if(uxQueueMessagesWaiting(Q3) != 0 ){

            if (xQueueReceive(Q3, &current\_task\_index, portMAX\_DELAY) == pdPASS) {

                printf("Q3 runs task%d! \r\n", current\_task\_index + 1);

                if (tasks[current\_task\_index].execution\_time > 0) {

                    vTaskResume(tasks[current\_task\_index].task\_handle);

                    temp = tasks[current\_task\_index].execution\_time - Q3\_TIME\_SLICES \* TIME\_SLICE\_INTERVAL;

                    if(temp > 0){

                        if(xQueueSendToBack(Q2, &current\_task\_index, portMAX\_DELAY) == pdPASS){

                            printf("task%d will be down to Q2. \r\n", current\_task\_index + 1);

                        }

                    }

                }

                vTaskDelay(Q3\_TIME\_SLICES \* TIME\_SLICE\_INTERVAL);

            }

        }else if(uxQueueMessagesWaiting(Q2) != 0){

            if(xQueueReceive(Q2, &current\_task\_index, portMAX\_DELAY) == pdPASS){

                printf("Q2 runs task%d! \r\n", current\_task\_index + 1);

                if (tasks[current\_task\_index].execution\_time > 0) {

                    vTaskResume(tasks[current\_task\_index].task\_handle);

                    temp = tasks[current\_task\_index].execution\_time - Q2\_TIME\_SLICES \* TIME\_SLICE\_INTERVAL;

                    if(temp > 0){

                        if(xQueueSendToBack(Q1, &current\_task\_index, portMAX\_DELAY) == pdPASS){

                            printf("task%d will be down to Q1. \r\n", current\_task\_index + 1);

                        }

                    }

                }

                vTaskDelay(Q2\_TIME\_SLICES \* TIME\_SLICE\_INTERVAL);

            }

        }else if(uxQueueMessagesWaiting(Q1) != 0){

            if(xQueueReceive(Q1, &current\_task\_index, portMAX\_DELAY) == pdPASS){

                printf("Q1 runs task%d! \r\n", current\_task\_index + 1);

                if (tasks[current\_task\_index].execution\_time > 0) {

                    vTaskResume(tasks[current\_task\_index].task\_handle);

                    temp = tasks[current\_task\_index].execution\_time - Q1\_TIME\_SLICES \* TIME\_SLICE\_INTERVAL;

                    if(temp > 0){

                        if(xQueueSendToBack(Q1, &current\_task\_index, portMAX\_DELAY) == pdPASS){

                            printf("task%d in Q1 again. \r\n", current\_task\_index + 1);

                        }

                    }

                }

                vTaskDelay(Q1\_TIME\_SLICES \* TIME\_SLICE\_INTERVAL);

            }

        }else{

            printf("No task in queue! \r\n");

            current\_time += TIME\_SLICE\_INTERVAL;

            show\_current\_time\_on\_lcd();

            vTaskDelay(TIME\_SLICE\_INTERVAL);

        }

    }

}

# 应用场景