

练习 9-1

要求：参考 `code/os/04-multitask`，在此基础上进一步改进任务管理功能。具体要求：

- 改进 `task_create()`，提供更多的参数。具体改进后的函数如下所示：

```
int task_create(void (*task)(void* param),
               void *param,
               uint8_t priority);
```

其中：

- `param` 用于在创建任务执行函数时可带入参数。如果没有参数则传入 `NULL`。
- `priority` 用于指定任务的优先级。目前要求最多支持 256 级，0 最高，依次类推。同时修改任务调度算法，在原先简单轮转的基础上支持按照优先级排序，优先选择优先级高的任务运行，同一级多个任务再轮转。
- 增加任务退出接口 `task_exit()`，当前任务可以通过调用该接口退出执行，内核负责将该任务回收，并调度下一个可运行任务。建议的接口函数如下：

```
void task_exit(void);
```

為了能夠記錄每個 task 的優先級，我新增了一個陣列

```
uint8_t context_priority[MAX_TASKS];
```

在 `os.h` 中，我新增了一個 `context` 用來記錄 kernel 進入 `schedule` 前的狀態，以便 task 結束後能返回 kernel。

```
struct context os_tasks;
```

接著我在 `kernel` 裡，`schedule()` 的前一步使用 `sys_switch()` 將當前 kernel 的狀態保存下來

```
printf("\n\n\n----- schedule start ----- \n\n\n");
//利用 sys_switch() 的特性，讓進入 schedule() 前的狀態能保存在 kernel_context 裡
sys_switch(&kernel_context, &kernel_context);
schedule();
```

製作 `task_exit()` 函數，呼叫它就可以利用 `switch_to()` 回到 kernel，並且剛好下一步就會再進入 `schedule()` 中

```
void task_exit()
{
    switch_to(&kernel_context);
}
```

另外我做了 10 個顯示不同圖案的 task 用來測試，他們會讀入一個字串指標的參數並顯示出來，以下是其中之一

```
//第 1 個 task
void user_task1(char* sentence)
{
    printf("Task 1: Running...|s\n", sentence);
    uart_puts("          |...\n");
    uart_puts("          |...\n");
    uart_puts("          |...\n");
    task_delay(6000);
    task_exit();
}
```

以下是我改寫的 task_create() 函數，在新增 task 時會把要給該 task 的參數放在它的 context 中的 a0 暫存器，並且記錄下它的優先級。

```
int task_create(void (*start_routine)(char *param), char *param, uint8_t priority)
{
    if (_top < MAX_TASKS) {
        //以當前的 TOP 值來分配給函數指標指向的 task 一個 context
        //並設定好它的 stack point 和 return address
        ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE - 1];
        ctx_tasks[_top].ra = (reg_t) start_routine;
        //將要給該 task 的參數設在該 context 的 a0 暫存器，這樣 context_switch 完就能使用
        ctx_tasks[_top].a0 = (reg_t) param;
        //把該 task 的 priority 存在對應的陣列位置中
        context_priority[_top] = priority;
        _top++;
        return 0;
    } else {
        return -1;
    }
}
```

另外也做一個用來存放 task 是否做過的陣列，做過就將值設為 1

```
uint8_t task_finish[MAX_TASKS];
```

有了以上的材料，下一頁開始實作 schedule()

實作 `schedule()` 時，我順便做了用來顯示當前 task 狀態的表

```
void schedule()
{
    int max = 0;
    int temp = 0;
    int final = 0;

    printf("\n|   task   ");

    for(int i = 0; i < MAX_TASKS; i++)
    {
        //顯示所有 task 編號
        printf("| task %d ", i);
    }
    printf("\n-----");

    printf("\n| priority ");
    //掃描整個 context_priority[] 陣列
    for(int i = 0; i < MAX_TASKS; i++)
    {
        //顯示所有 task 的優先級
        printf("|   %d   ", context_priority[i]);
        //找到優先度最高的 task，並且必須是沒有被做過的
        if( context_priority[i] >= temp && task_finish[i] != 1)
        {
            temp = context_priority[i];
            //將最優先要做的 task 編號設給 max 變數
            max = i;
        }
    }
    printf("\n-----");

    printf("\n|   finish   ");
    for(int i = 0; i < MAX_TASKS; i++)
    {
        //顯示所有 task 是否被執行過的狀態
        printf("|   %d   ", task_finish[i]);
    }
    printf("\n\n");

    //找到優先度最低的 task
    for(int i = 0; i < MAX_TASKS; i++)
    {
```

```

    if( context_priority[i] <= temp )
    {
        temp = context_priority[i];
        //將最低的 task 編號設給 final 變數
        final = i;
    }
}

//如果 context_priority[final]為0 且 task_finish[final] 也是0，代表所有任務都已經輪過一遍
if(context_priority[final] == 0 && task_finish[final] == 1)
{
    //將所有優先度不為 0 的 task 的完成狀態都重設為 0
    for(int i = 0; i< MAX_TASKS; i++)
    {
        if(context_priority != 0)
        {
            task_finish[i] = 0 ;
        }
    }
    printf("***** task
reset *****\n");
}
//不是的話代表還有 task 沒做完，於是將下個 task 的 context 指針設為 max 所代表的 task 然後
switch_to()
else
{
    printf("max = task %d\n\n\n", max);
    struct context *next = &ctx_tasks[max];
    task_finish[max] = 1; //將該 task 的 task_finish 設為 1，代表已經做了
    switch_to(next);
}
}

```

編譯執行看看

第一個執行的是 task3 (此處我以數字大的為優先，因為用 0 來做為最後一個執行的優先度程式碼會比較簡潔)

```
----- schedule start -----  
  
      task | task 0 | task 1 | task 2 | task 3 | task 4 | task 5 | task 6 | task 7 | task 8 | task 9 |  
-----  
priority | 2     | 3     | 7     | 9     | 5     | 4     | 6     | 7     | 8     | 0     |  
-----  
finish   | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |  
  
max = task 3  
  
Task 3: Running.....|.....d  
              ...|  
              ...|  
              ...|
```

第二個執行的是 task8，同時可以看到 task3 的 finish 被標為 1 了

```
      task | task 0 | task 1 | task 2 | task 3 | task 4 | task 5 | task 6 | task 7 | task 8 | task 9 |  
-----  
priority | 2     | 3     | 7     | 9     | 5     | 4     | 6     | 7     | 8     | 0     |  
-----  
finish   | 0     | 0     | 0     | 1     | 0     | 0     | 0     | 0     | 0     | 0     |  
  
max = task 8  
  
Task 8: Running...|i  
              |...  
              |...  
              |...
```

以下依序

```
      task | task 0 | task 1 | task 2 | task 3 | task 4 | task 5 | task 6 | task 7 | task 8 | task 9 |  
-----  
priority | 2     | 3     | 7     | 9     | 5     | 4     | 6     | 7     | 8     | 0     |  
-----  
finish   | 0     | 0     | 0     | 1     | 0     | 0     | 0     | 0     | 1     | 0     |  
  
max = task 7  
  
Task 7: Running.....|.....h  
              ...|  
              ...|  
              ...|
```

如果兩個 task 同級，因為用掃描搜尋陣列的特性，在陣列後方的會先做，也就是 task7 -> task2

後
↓

先
↖

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	2	3	7	9	5	4	6	7	8	0
finish	0	0	0	1	0	0	0	1	1	0

max = task 2

Task 2: Running...|c
|...
|...
|...

↓

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	2	3	7	9	5	4	6	7	8	0
finish	0	0	1	1	0	0	0	1	1	0

max = task 6

Task 6: Running...|g
|...
|...
|...

↓

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	2	3	7	9	5	4	6	7	8	0
finish	0	0	1	1	0	0	1	1	1	0

max = task 4

Task 4: Running...|e
|...
|...
|...

↓

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	2	3	7	9	5	4	6	7	8	0
finish	0	0	1	1	1	1	1	1	1	0

max = task 1

Task 1: Running.....|.....b
 ...|
 ...|
 ...|

↓

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	2	3	7	9	5	4	6	7	8	0
finish	0	1	1	1	1	1	1	1	1	0

max = task 0

Task 0: Running...|a
 |...
 |...
 |...

↓

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	2	3	7	9	5	4	6	7	8	0
finish	1	1	1	1	1	1	1	1	1	0

max = task 9

Task 9: Running.....|.....j
 ...|
 ...|
 ...|

全部任務完成後便重設所有 finish 狀態。為了測試，這個 schedule() 還沒有使用無限循環，所以會離開並進到 kernel 裡的下一個程式。

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	2	3	7	9	5	4	6	7	8	0
finish	1	1	1	1	1	1	1	1	1	1

***** task reset *****

----- uart polling input -----
 please input , and use Enter to newline , or use exit to quit

练习 9-2

目前 `code/os/04-multitask` 实现的任务调度中，前一个用户任务直接调用 `task_yield()` 函数并最终调用 `switch_to()` 切换到下一个用户任务。`task_yield()` 作为内核路径借用了用户任务的栈，当用户任务的函数调用层次过多或者 `task_yield()` 本身函数内部继续调用函数，可能会导致用户任务的栈空间溢出。参考 "mini-riscv-os" 的 `03-MultiTasking` 的实现，为内核调度单独实现一个任务，在任务切换中，前一个用户任务首先切换到内核调度任务，然后再由内核调度任务切换到下一个用户任务，这样就可以避免前面提到的问题了。

要求：参考以上设计，并尝试实现之。

用 9-1 的作業來延伸，這次選擇用 `task0` 來當作內核調度任務

```
//將所有 task 初始化
void schedule_init()
{
    //優先把 schedule() 設為 task0，每個 task 離開後都會先進 schedule()
    ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE - 1];
    ctx_tasks[_top].ra = (reg_t) schedule;
    _top++;
    task_create(user_task1, "a", 2);
    task_create(user_task2, "b", 3);
    task_create(user_task3, "c", 7);
    task_create(user_task4, "d", 9);
    task_create(user_task5, "e", 5);
    task_create(user_task6, "f", 4);
    task_create(user_task7, "g", 6);
    task_create(user_task8, "h", 7);
    task_create(user_task9, "i", 8);
}
```

`Task_exit()` 用 `switch_to` 跳回 `schedule()`

```
void task_exit()
{
    switch_to( &ctx_tasks[0]);
}
```

下一頁是 `schedule()`


```

void schedule()
{
    int max = 0;
    int temp = 0;
    printf("\n|   task   ");
    for(int i = 0; i< MAX_TASKS; i++)
    {
        //顯示所有 task 編號
        printf("| task %d ", i);
    }
    printf("\n-----");
    printf("\n| priority ");
    //掃描整個 context_priority[] 陣列
    for(int i = 0; i< MAX_TASKS; i++)
    {
        //顯示所有 task 的優先級
        printf("|   %d   ", context_priority[i]);
        //找到優先度最高的 task，並且必須是沒有被做過的
        if( context_priority[i] >= temp && task_finish[i] != 1)
        {
            //將最優先要做的 task 編號設給 max 變數
            temp = context_priority[i];
            max = i;
        }
    }
    printf("\n-----");
    printf("\n|   finish   ");
    for(int i = 0; i< MAX_TASKS; i++)
    {
        //顯示所有 task 是否被執行過的狀態
        printf("|   %d   ", task_finish[i]);
    }
    printf("\n\n");

    //如果 context_priority[max]為 0 且 task_finish[max] 也是 0，代表所有任務都已經輪過一遍
    if(context_priority[max] == 0 && task_finish[max] == 0)
    {
        for(int i = 0; i< MAX_TASKS; i++)
        {

```

```

        //將所有優先度不為 0 的 task 的完成狀態都重設為 0
        if(context_priority != 0)
        {
            task_finish[i] = 0 ;
        }
    }

    printf("***** task
reset *****\n");
}

//不是的話代表還有 task 沒做完，於是將下個 task 的 context 指針設為 max 所代表的 task 然後
switch_to()
else
{
    printf("max = task %d\n\n\n", max);
    struct context *next = &ctx_tasks[max];
    //將該 task 的 task_finish 設為 1，代表已經做了
    task_finish[max] = 1;
    switch_to(next);
}
}

```

可以看到，這次的 `schedule()` 更優雅，不需要檢查所有的任務是否做完，最後的任務一定是 `task0`，這時候就能輕易判斷並重設所有狀態。而且這時的 `context`，`ra` 暫存器內的地址就是 `schedule()` 本身，所以重設完離開 `schedule()` 後又自動跳回 `schedule()` 開始下一個輪迴，因此不需要在 `schedule()` 內使用無限循環。

下面的圖可以看到重設完後又自動進入新的輪迴

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	0	2	3	7	9	5	4	6	7	8
finish	0	1	1	1	1	1	1	1	1	1

***** task reset *****

task	task 0	task 1	task 2	task 3	task 4	task 5	task 6	task 7	task 8	task 9
priority	0	2	3	7	9	5	4	6	7	8
finish	0	0	0	0	0	0	0	0	0	0

max = task 4

Task 4: Running.....|.....d

...|

...|

...|