

实验报告

【实验题目】

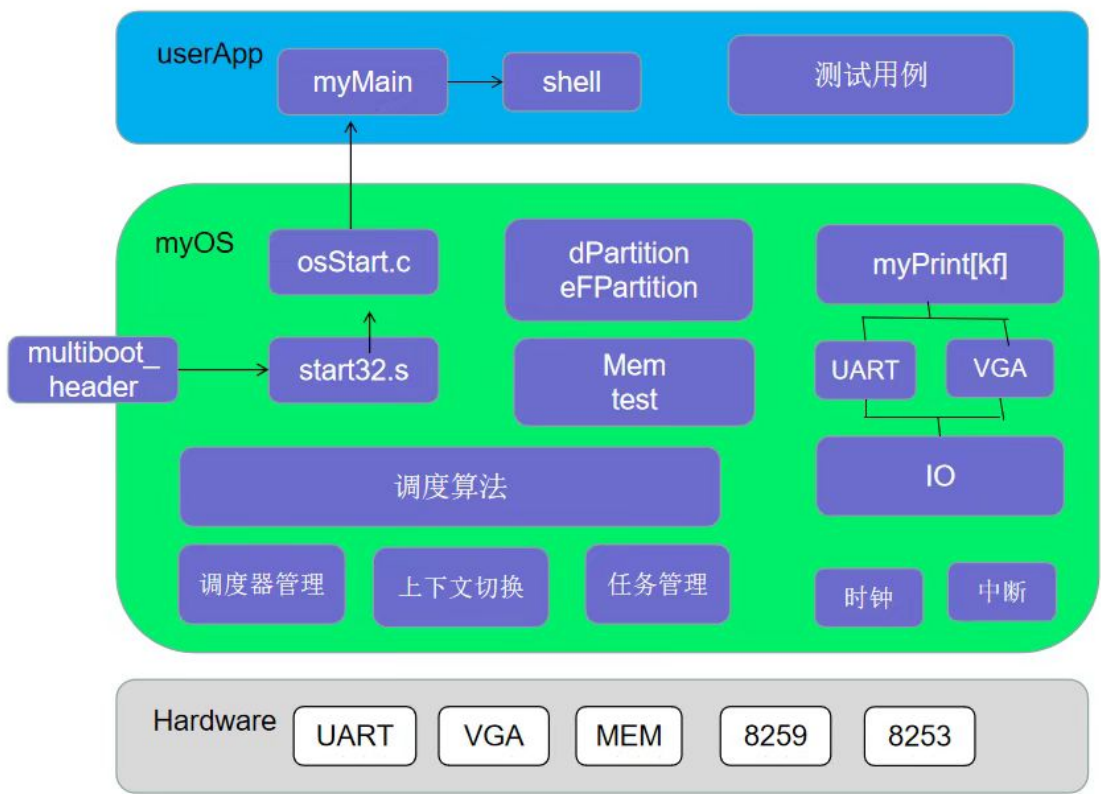
任务调度

【实验要求】

- 1. 完成任务控制块以及就绪队列的数据结构
- 2. 完成任务创建、销毁、启动、结束原语
- 3. 实现任务动态到达, 完成上下文切换
- 4. 实现多种调度算法的调度器(至少一种是抢占式调度算法, 调度器和任务参数采用统一接口)
- 5. 完成任务管理器的初始化, 可以正常进入多任务运行状态

【实验原理】

- 软件架构和功能



软件整个生命周期涉及三个层次：硬件层，操作系统层，用户应用层，其功能分别如下：

1. Hardware层：
 - ① UART和VGA (IO功能)
 - ② PIC i8259 (可中断控制器的中断处理功能)
 - ③ PIT i8253 (可编程间隔定时器的定时功能)
 - ④ Memory (内存空间)
2. myOS层：
 - ① 各种硬件的初始化及处理工作 (IDT的初始化及中断处理程序, 定时器的时钟显示模块)
 - ② 硬件功能的接口, 系统调用 (UART和VGA的IO接口函数等)
 - ③ 支持C程序的库函数 (myPrintf, Malloc等)

④ 上层软件运行环境的初始化,主要为C语言程序 (栈的建立以及BSS段的0初始化,为C程序的执行提供运行环境)

⑤ 上层软件运行时内存,CPU时间分配管理 (内存空间的管理与分配,多任务调度管理模块)

3. UserApp层:

① 实现shell终端:接受相关指令输入并执行

② 内存管理与分配测试程序

③ 任务管理测试用例

• 软件执行流程

Multiboot header(启动) ==> start32.S(准备上下文) ==> osStart.c(初始化操作系统环境,初始化任务管理器并进入多任务处理,以myMain为初始进程) ==> main.c(按照任务调度进入myMain进程并创建其他进程) ==> 操作系统持续调度中 ==> 测试任务执行结束,陷入Idle默认任务

本实验的主流程如上:

1. 在 multiboot_header 中完成系统的启动
2. 通过start32.S完成系统初始化并为C语言准备环境,并调用osStart.c进入C程序
3. 在 osStart.c中完成初始化8259A,初始化8253,清屏及内存初始化等操作,再初始化任务调度器(其中以创建myMain为第一个进程,另外创建idle任务作为默认任务),之后进入多任务调度
4. 运行 myMain 中的代码,通过控制任务到达的时间,创建并启动多个进程,完成进程调度的测试,完成后以同样方式创建并启动shell进程,等待调度
5. 在完成进程调度测试后,按调度进入shell程序,等待与用户交互
6. 测试任务执行结束,调度器执行默认的idle任务

• 新增功能模块及其实现

本次实验主要为实现任务调度的各种数据结构,调度器,任务创建、销毁原语,任务队列,任务动态到达,以及任务管理的统一接口.具体介绍如下:

◦ 使用的数据结构

本次实验主要使用了:

程序控制块(包含任务调度的相关信息以及进程的状态,拥有的资源)

hook函数链节点 (实现时钟中断函数调用链)

任务队列 (包含FIFO和Prio两种任务队列,用于不同调度器的就绪队列,以及任务到达队列)

调度器 (包含任务调度所需的各种功能接口,实现了统一接口)

任务管理器(该数据结构整合了操作系统任务管理的所有数据结构,统一管理任务调度的各种资源,负责任务管理模块的所有功能)

```
#define MaxTaskNum      40+2
#define STACK_SIZE      4096

#define MAX_PRIORITY     5
#define MAX_EXEC_TIME    20

#define PRIORITY         0
#define EXEC_TIME        1
#define ARRIV_TIME       2

#define FCFS              1
#define PRIO              2
```

```

#define      SJF      3

// 进程状态
typedef enum {
    T_UNALLOC, T_READY, T_RUN, T_WAIT
} TSKSTAT;

// 进程调度器所需的任务参数
typedef struct tskPara{
    int priority;
    int exeTime;
    int arrTime;
} tskPara;

// hook函数链结点
typedef struct hookNode{
    void (*hook_func)(void);
    struct hookNode* next;
}hookNode;

// hook函数链
typedef struct hookList {
    hookNode* head;
    hookNode* tail;
    int num;
}hookList;

// 程序控制块
typedef struct myTCB{
    int          tid;
    TSKSTAT      state;
    unsigned int  runTime;
    unsigned int  lastScheduledTime;
    tskPara*     para;
    void (*tskBody)(void);
    unsigned long*      stack;
    unsigned long*      topOfStack;
    struct myTCB*  next;
} myTCB;

// 任务队列节点
typedef struct taskQueueNode {
    myTCB *TCB;
    struct taskQueueNode *next;
} taskQueueNode;

// 先进先出任务队列
typedef struct taskQueueFIFO {
    taskQueueNode *head,*tail;
} taskQueueFIFO;

// 优先任务队列
typedef struct taskQueuePrio {
    taskQueueNode *head,*tail;
    int          capacity;
    int          (*cmp)(const myTCB *a, const myTCB *b);
} taskQueuePrio;

```

```
// 调度器
typedef struct scheduler {
    unsigned int type;
    unsigned int preempt;
    myTCB* (*nextTsk)(void);
    void (*enqueueTsk)(myTCB* tsk);
    void (*dequeueTsk)(int tid);
    void (*schedulerInit)(void);
    void (*schedule)(void);
    void (*tick_hook)(void);
} scheduler;

// 任务管理器
typedef struct TaskCtl{
    myTCB tcbPool[MaxTaskNum];
    scheduler sched;
    taskQueueFIFO *rdyQueueFIFO;
    taskQueuePrio * rdyQueuePRIO;
    taskQueuePrio * arrvQueue;
    myTCB* idleTsk;
    myTCB* nowTsk;
    myTCB* firstFreeTsk;
    int taskNum;
}TaskCtl;
```

调度器

本次实验依托三种调度器, 实现了任务调度的三种算法(FCFS, SJF, PRIO), 其中PRIO为抢占式调度算法, 调度器封装为统一接口

```
#include "../include/task.h"
#include "../include/tick.h"
#include "../include/myPrintk.h"

// 调度器初始化
void schedulerInit(void) {
    switch (taskctl.sched.type) {
        case FCFS:
            taskctl.sched.schedulerInit = schedulerInitFCFS;
            taskctl.sched.preempt = 0;
            taskctl.sched.nextTsk = nextTskFCFS;
            taskctl.sched.enqueueTsk = enqueueTskFCFS;
            taskctl.sched.dequeueTsk = dequeueTskFCFS;
            taskctl.sched.schedule = scheduleFCFS;
            taskctl.sched.tick_hook = 0;
            break;

        case PRIO:
            taskctl.sched.schedulerInit = schedulerInitPrio;
            taskctl.sched.preempt = 1;
            taskctl.sched.nextTsk = nextTskPrio;
            taskctl.sched.enqueueTsk = enqueueTskPrio;
            taskctl.sched.dequeueTsk = dequeueTskPrio;
            taskctl.sched.schedule = schedulePrio;
            taskctl.sched.tick_hook = 0;
    }
}
```

```

        break;
    case SJF:
        taskctl.sched.schedulerInit = schedulerInitSJF;
        taskctl.sched.preempt = 0;
        taskctl.sched.nextTsk = nextTskSJF;
        taskctl.sched.enqueueTsk = enqueueTskSJF;
        taskctl.sched.dequeueTsk = dequeueTskSJF;
        taskctl.sched.schedule = scheduleSJF;
        taskctl.sched.tick_hook = 0;
        break;
    }
    taskctl.sched.schedulerInit();
}

/* ***** FCFS调度器 ***** */
***** */

myTCB * nextTskFCFS(void) {
    return taskQueueFIFONext(taskctl.rdyQueueFIFO);
}

void enqueueTskFCFS(myTCB *tsk) {
    taskQueueFIFOEnqueue(taskctl.rdyQueueFIFO, tsk);
}

void dequeueTskFCFS(int tid) {
    taskQueueFIFODequeue(taskctl.rdyQueueFIFO, tid);
}

void schedulerInitFCFS(void) {
    taskQueueFIFOInit(taskctl.rdyQueueFIFO);
}

void scheduleFCFS(void) {
    taskQueueFIFO * rdyQueueFCFS = taskctl.rdyQueueFIFO;
    while (1) {

        myTCB *preTsk = taskctl.nowTsk;

        if (taskQueueFIFOEmpty(rdyQueueFCFS)) {
            taskctl.nowTsk = taskctl.idleTsk;
        }
        else{
            taskctl.nowTsk = taskQueueFIFONext(rdyQueueFCFS);
            myPrintk(0x05, "Tasknow id is %d\n", taskctl.nowTsk->tid);
        }
        if (taskctl.nowTsk == taskctl.idleTsk)
            continue;
        taskctl.nowTsk->state = T_RUN;
        context_switch(&BspContext, taskctl.nowTsk->topOfStack);
    }
}

/* ***** SJF调度器 ***** */
***** */

int compare_exec_time(const myTCB *a, const myTCB *b) {

```

```

        if (getTskPara(EXEC_TIME, a->para) == getTskPara(EXEC_TIME, b->para)){
            return getTskPara(ARRV_TIME, a->para) - getTskPara(ARRV_TIME, b->para);
        }
        else{
            return getTskPara(EXEC_TIME, a->para) - getTskPara(EXEC_TIME, b->para);
        }
    }

myTCB * nextTskSJF(void) {
    return taskQueuePrioNext(taskctl.rdyQueuePRIO);
}

void enqueueTskSJF(myTCB *tsk) {
    taskQueuePrioEnqueue(taskctl.rdyQueuePRIO, tsk);
}

void dequeueTskSJF(int tid) {
    taskQueuePrioDequeue(taskctl.rdyQueuePRIO, tid);
}

void schedulerInitSJF(void) {
    taskQueuePrioInit(taskctl.rdyQueuePRIO, taskctl.taskNum,
compare_exec_time);
}

void scheduleSJF(void) {
    taskQueuePrio * rdyQueueSJF = taskctl.rdyQueuePRIO;
    while (1) {
        myTCB *preTsk = taskctl.nowTsk;

        if (taskQueuePrioEmpty(rdyQueueSJF)) {
            taskctl.nowTsk = taskctl.idleTsk;
        }
        else{
            taskctl.nowTsk = taskQueuePrioNext(rdyQueueSJF);
            myPrintk(0x05, "Tasknow id is %d\n", taskctl.nowTsk->tid);
        }

        if (preTsk == taskctl.idleTsk && taskctl.nowTsk ==
taskctl.idleTsk)
            continue;

        taskctl.nowTsk->state = T_RUN;
        context_switch(&BspContext, taskctl.nowTsk->topOfStack);
    }
}

/*****Prio调度器*****/

int compare_priority(const myTCB *a, const myTCB *b) {
    if (getTskPara(PRIORITY, a->para) == getTskPara(PRIORITY, b->para))

```

```

        return getTskPara(ARRV_TIME, a->para) - getTskPara(ARRV_TIME, b->para);
    else
        return getTskPara(PRIORITY, a->para) - getTskPara(PRIORITY, b->para);
    }

myTCB * nextTskPrio(void) {
    return taskQueuePrioNext(taskctl.rdyQueuePRIO);
}

void enqueueTskPrio(myTCB *tsk) {
    taskQueuePrioEnqueue(taskctl.rdyQueuePRIO, tsk);
}

void dequeueTskPrio(int tid) {
    taskQueuePrioDequeue(taskctl.rdyQueuePRIO, tid);
}

void schedulerInitPrio(void) {
    taskQueuePrioInit(taskctl.rdyQueuePRIO, taskctl.taskNum,
compare_priority);
}

void schedulePrio(void) {

    taskQueuePrio* rdyQueuePrio = taskctl.rdyQueuePRIO;

    while (1) {
        myTCB *preTsk = taskctl.nowTsk;
        if (taskQueuePrioEmpty(rdyQueuePrio)) {
            taskctl.nowTsk = taskctl.idleTsk;
        }
        else{
            taskctl.nowTsk = taskQueuePrioNext(rdyQueuePrio);
            myPrintk(0x05, "Tasknow id is %d\n", taskctl.nowTsk->tid);
        }
        if (preTsk == taskctl.idleTsk && taskctl.nowTsk ==
taskctl.idleTsk)
            continue;
        taskctl.nowTsk->state = T_RUN;
        context_switch(&BspContext, taskctl.nowTsk->topOfStack);
    }
}

```

○ 任务调度原语、

本次实验依托具体各种调度器实现任务创建、销毁、启动、结束的各种原语,其中任务启动可依据当前任务控制器采取的调度策略决定是否抢占调度

```

#include "../include/task.h"
#include "../include/malloc.h"
#include "../include/tick.h"
#include "../include/myPrintk.h"
#include "../include/interrupt.h"

```

```

// 任务启动(可依据当前任务控制器采取的调度策略决定是否抢占调度)
void tskStart(int tid){
    taskctl.tcbPool[tid].state = T_READY;
    taskctl.sched.enqueueTsk(&taskctl.tcbPool[tid]);
    if(taskctl.sched.preempt &&taskctl.nowTsk &&
taskctl.sched.nextTsk()){
        if(taskctl.sched.nextTsk() != taskctl.nowTsk){
            context_switch(&taskctl.nowTsk->topOfStack, BspContext);
        }
    }
}

// 任务结束
void tskEnd(void){
    taskctl.sched.dequeueTsk(taskctl.nowTsk->tid);
    destroyTsk(taskctl.nowTsk->tid);
    myPrintk(0x06, "task end! tid :%d\n", taskctl.nowTsk->tid);
    context_switch(&taskctl.nowTsk->topOfStack, BspContext);
}

// 任务创建
int createTsk(void (*tskBody)(void)){

    myTCB* tcb = taskctl.firstFreeTsk;
    if(tcb == 0)
        return -1;
    taskctl.firstFreeTsk = taskctl.firstFreeTsk->next;
    tcb->next = 0;

    initTskPara(&tcb->para);

    tcb->tskBody = tskBody;

    stack_init(&(tcb->topOfStack), tskBody);

    return tcb->tid;
}

// 任务销毁
void destroyTsk(int tid){
    taskctl.tcbPool[tid].state = T_UNALLOC;
    taskctl.tcbPool[tid].runTime = 0;
    taskctl.tcbPool[tid].lastScheduledTime = 0;
    taskctl.tcbPool[tid].para = 0;
    taskctl.tcbPool[tid].tskBody = 0;
    taskctl.tcbPool[tid].next = taskctl.firstFreeTsk;
    taskctl.firstFreeTsk = &taskctl.tcbPool[tid];
}

```

o 任务队列

为任务调度、任务动态到达实现两种队列, 以及各种操作

```
#include "../include/task.h"
```



```

#include "../include/interrupt.h"
#include "../include/malloc.h"
#include "../include/myPrintk.h"

// FIFO队列
void taskQueueFIFOPrint(taskQueueFIFO *queue){
    if(!queue){
        myPrintk(0x08,"Error! FIFO queue is null\n");
    }
    else {
        taskQueueNode * node = queue->head;
        while(node){
            myPrintk(0x05,"node tid is %d\n",node->TCB->tid);
            node = node->next;
        }
    }
}

void taskQueueFIFOInit(taskQueueFIFO *queue){
    queue->head = 0;
    queue->tail = 0;
}

int taskQueueFIFOEmpty(taskQueueFIFO *queue){
    return (queue->head == 0 );
}

myTCB * taskQueueFIFONext(taskQueueFIFO *queue){
    if (taskQueueFIFOEmpty(queue))
        return 0;
    else
        return queue->head->TCB;
}

void taskQueueFIFOEnqueue(taskQueueFIFO *queue, myTCB *tsk) {
    taskQueueNode *node = (taskQueueNode
*)malloc(sizeof(taskQueueNode));
    if (!node)
        return ;

    node->TCB = tsk;
    node->next = 0;

    disable_interrupt();

    if (taskQueueFIFOEmpty(queue))
        queue->head = node;
    else {
        queue->tail->next = node;
    }
    queue->tail = node;
    enable_interrupt();
}

void taskQueueFIFODequeue(taskQueueFIFO *queue,int tid) {
    taskQueueNode *node = queue->head;
    if (taskQueueFIFOEmpty(queue))
        return ;

```

```

        else {
            disable_interrupt();
            if(queue->head == queue->tail){
                queue->head = 0;
                queue->tail = 0;
            }
            else {
                queue->head = node->next;
            }
            enable_interrupt();
        }
    }
}

// Prio队列
void taskQueuePrioPrint(taskQueuePrio *queue){

    if(!queue){
        myPrintk(0x08,"Error! Prio queue is null\n");
    }
    else {
        taskQueueNode * node = queue->head;
        while(node){
            myPrintk(0x05,"node tid is %d\n",node->TCB->tid);
            node = node->next;
        }
    }
}

void taskQueuePrioInit(taskQueuePrio *queue, int capacity, int (*cmp)
(const myTCB *a, const myTCB *b)) {
    queue->head = 0;
    queue->tail = 0;
    queue->capacity = capacity;
    queue->cmp = cmp;
}

int taskQueuePrioEmpty(taskQueuePrio *queue) {
    return (queue->head == 0 && queue->tail == 0);
}

myTCB * taskQueuePrioNext(taskQueuePrio *queue) {
    if (taskQueuePrioEmpty(queue))
        return 0;
    else{
        return queue->head->TCB;
    }
}

void taskQueuePrioEnqueue(taskQueuePrio *queue, myTCB *tsk) {
    disable_interrupt();
    taskQueueNode* node = (taskQueueNode*)malloc(sizeof(taskQueueNode));
    node->TCB = tsk;
    node->next = 0;
}

```

```

taskQueueNode* p = queue->head;
taskQueueNode* q;
if(!queue->head){
    queue->head = node;
    queue->tail = node;
}
else if(queue->cmp(queue->head->TCB, tsk)>0){
    node->next = queue->head;
    queue->head = node;
}
else {
    q = p->next;
    while(q && queue->cmp(q->TCB, tsk)<=0){
        p = q;
        q = q->next;
    }
    if(!q){
        queue->tail->next = node;
        queue->tail = node;
    }
    else {
        node->next = q;
        p->next = node;
    }
}
enable_interrupt();
}

void taskQueuePrioDequeue(taskQueuePrio *queue,int tid) {
    disable_interrupt();
    taskQueueNode *p = queue->head;
    taskQueueNode *q;
    if(p->TCB->tid == tid){
        queue->head = p->next;
        if(p == queue->tail){
            queue->tail = 0;
        }
    }
    else {
        q = p->next;
        while(q->TCB->tid != tid){
            p = q;
            q = q->next;
        }
        p->next = q->next;
    }
    enable_interrupt();
}

```

- 任务动态到达
 - Hook机制

由于任务动态到达需要依托时钟中断,为实现可扩展性,实现一个时钟中断处理的函数调用链(支持动态扩展)

```
#include "../include/tick.h"
#include "../include/malloc.h"
#include "../include/myPrintk.h"

int tick_number = 0;

void startArrivedTask_hook(void);
void oneTickUpdateWallClock(void);

// 时钟中断函数调用链初始化
void tick_hook_init(void){
    tick_hook_list.head = (hookNode*)malloc(sizeof(hookNode));
    tick_hook_list.head->hook_func = oneTickUpdateWallClock;
    tick_hook_list.head->next = 0;
    tick_hook_list.tail = tick_hook_list.head;
    tick_hook_list.num = 1;
}

void print_tick_list(void){
    hookNode* node = tick_hook_list.head;
    myPrintk(0x09, "hooklist length %d\n", tick_hook_list.num);
    while(node){
        myPrintk(0x08, "hooklist: %d\n", node->hook_func);
        node = node->next;
    }
}

// 时钟中断处理函数
void tick(void){

    tick_number++;

    hookNode* hook_ptr = tick_hook_list.head;

    while(hook_ptr){
        hook_ptr->hook_func();
        hook_ptr = hook_ptr->next;
    }
    return ;
}

int get_tick_number(void){
    return tick_number;
}

// 拓展函数调用链
void append2HookList(hookList* hooklist, void(*func)(void)){
    hookNode* node = (hookNode*)malloc(sizeof(hookNode));
    node->hook_func = func;
    node->next = 0;
    hooklist->tail->next = node;
    hooklist->tail = node;
    hooklist->num ++;
}
```

```
}
```

■ 任务到达队列

依托任务优先队列实现任务到达队列,同时向时钟中断函数调用链中添加钩子函数,实现任务按照到达时间加入就绪队列

```
#include "../include/task.h"
#include "../include/tick.h"
#include "../include/myPrintk.h"

taskQueuePrio arrvQueue;

int compare_arrv_time(const myTCB *a, const myTCB *b) {
    return getTskPara(ARRV_TIME, a->para) - getTskPara(ARRV_TIME, b->para);
}

// 实现任务动态到达
void startArrivedTask_hook(void) {
    if (taskQueuePrioEmpty(taskctl.arrvQueue))
        return;

    myTCB *nextTsk = taskQueuePrioNext(taskctl.arrvQueue);

    if (get_time() >= getTskPara(ARRV_TIME, nextTsk->para)) {
        myPrintk(0x02, "Now timer is %d, Tid %d arrive\n", get_time(), nextTsk->tid);
        taskQueuePrioDequeue(taskctl.arrvQueue, nextTsk->tid);
        tskStart(nextTsk->tid);
    }
    return ;
}

// 任务到达队列初始化
void taskArrvQueueInit(void) {
    taskctl.arrvQueue = &arrvQueue;
    taskQueuePrioInit(taskctl.arrvQueue, 0, compare_arrv_time);
    append2HookList(&tick_hook_list, startArrivedTask_hook);
}

// 任务加入到达队列
void enableTask(int tid) {
    if (taskctl.tcbPool[tid].para->arrTime == 0){
        myPrintk(0x02, "Now timer is %d, Tid %d arrive\n", get_time(), tid);
        tskStart(tid);
    }
    else{
        taskQueuePrioEnqueue(taskctl.arrvQueue, &taskctl.tcbPool[tid]);
    }
}
```

○ 任务管理器

本次实验实现任务调度模块的封装, 将所有任务调度处理的相关数据结构以及资源封装进任务管理器, 使得任务管理更具模块性同时方便扩展, 所有的任务管理统一由任务管理器负责

// 任务管理器初始化

```
void TaskCtlInit(void){

    myTCB* tcb;

    for(int i=0;i<MaxTaskNum;i++){
        tcb = &taskctl.tcbPool[i];
        tcb->tid = i;
        tcb->state = T_UNALLOC;
        tcb->runTime = 0;
        tcb->lastScheduledTime = 0;
        tcb->para = 0;
        tcb->tskBody = 0;
        tcb->stack = (unsigned long*)malloc(STACK_SIZE);
        tcb->topOfStack = tcb->stack + STACK_SIZE-1;
        tcb->next = (i== MaxTaskNum -1)? 0 : &taskctl.tcbPool[i+1];
    }

    taskctl.rdyQueueFIFO =
(taskQueueFIFO*)malloc(sizeof(taskQueueFIFO));
    taskctl.rdyQueuePRIO =
(taskQueuePrio*)malloc(sizeof(taskQueuePrio));

    taskctl.arrvQueue = 0;

    taskctl.nowTsk = 0;

    taskctl.idleTsk = &taskctl.tcbPool[0];
    taskctl.idleTsk->tskBody = tskIdleBody;
    taskctl.idleTsk->state = T_READY;
    taskctl.idleTsk->next = 0;
    stack_init(&(taskctl.idleTsk->topOfStack), tskIdleBody);

    taskctl.firstFreeTsk = &taskctl.tcbPool[1];

    schedulerInit();

    //创建并启动初始任务
    int initTid = createTsk(tskInitBody);
    setTskPara(Arrv_TIME, 0, taskctl.tcbPool[initTid].para);
    setTskPara(EXEC_TIME, 0, taskctl.tcbPool[initTid].para);
    setTskPara(PRIORITY, 0, taskctl.tcbPool[initTid].para);
    tskStart(initTid);
}

// 多任务调度启动
void startMultiTask(void) {
    BspContext = BspContextBase + STACK_SIZE - 1;
    taskctl.sched.schedule();
}

// 任务上下文切换
```

```

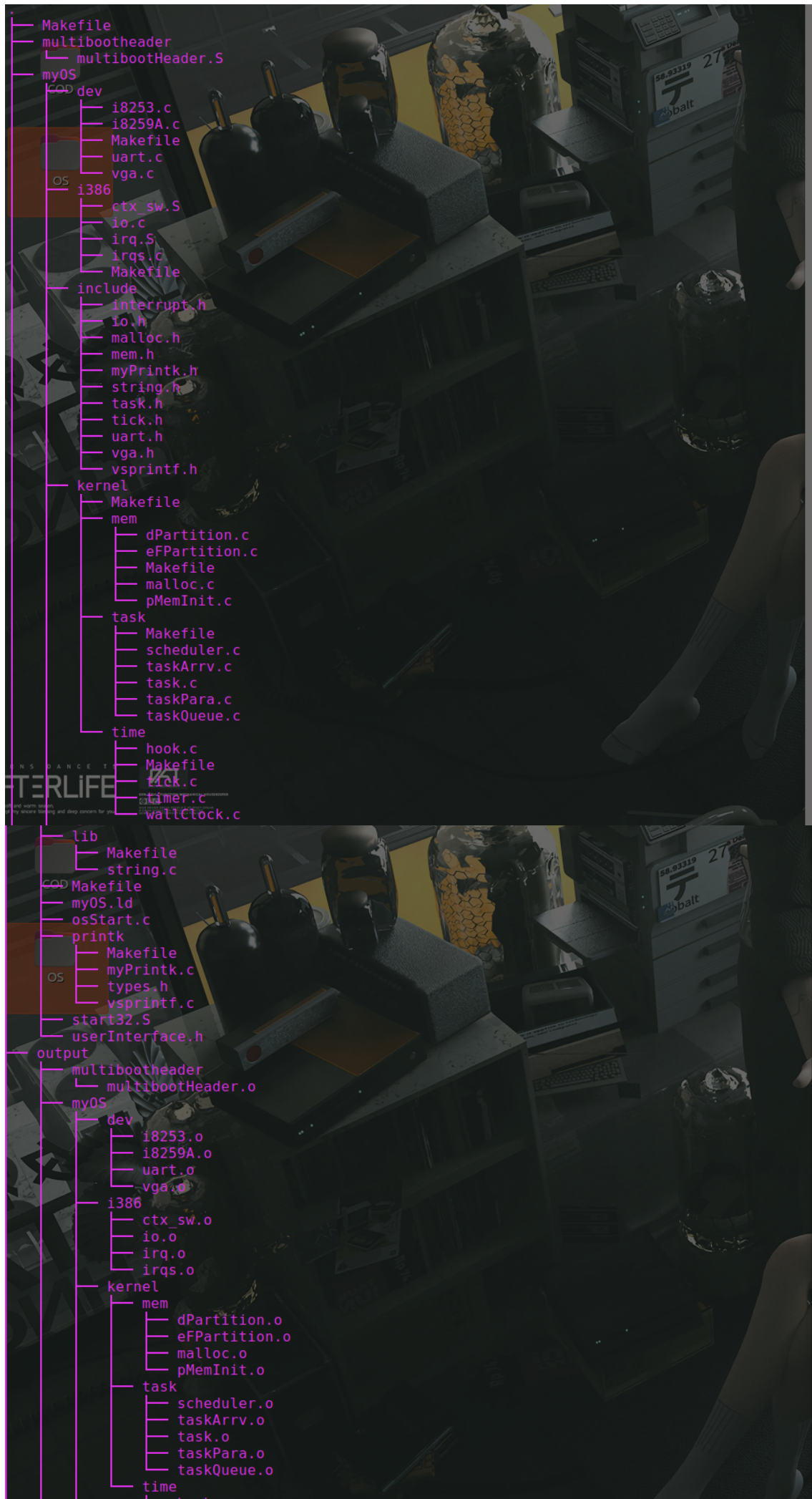
void context_switch(unsigned long **prevTskStkAddr, unsigned long
*nextTskStk) {
    prevTSK_StackPtrAddr = prevTskStkAddr;
    nextTSK_StackPtr = nextTskStk;
    CTX_SW();
}

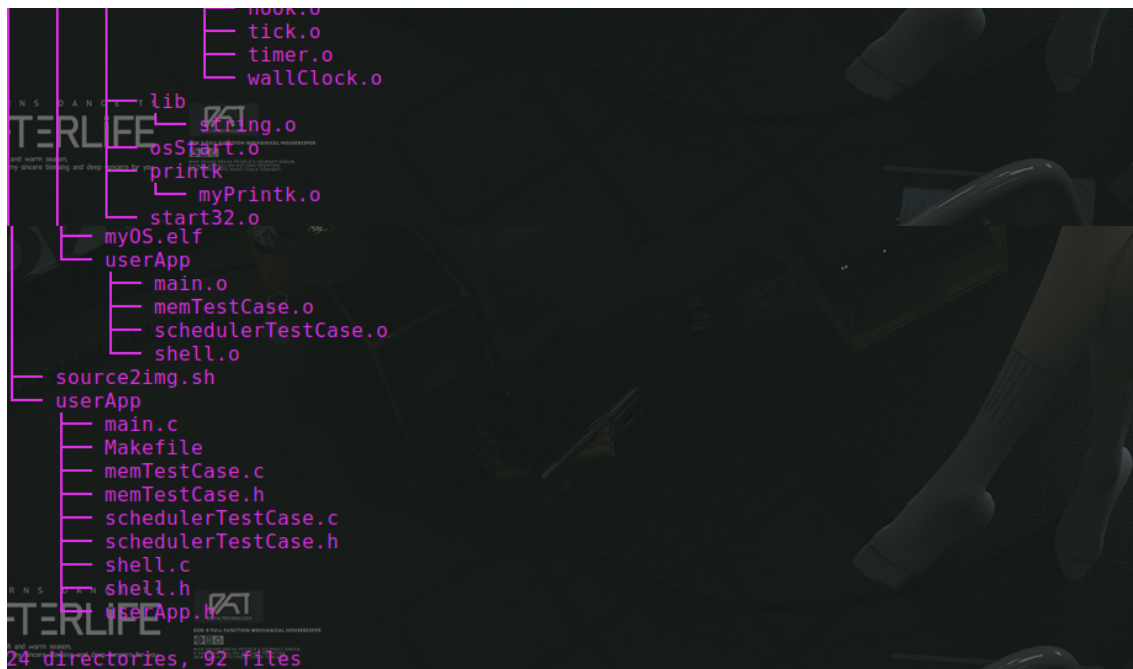
// 任务栈初始化
void stack_init(unsigned long **stk, void (*task)(void)) {
    *(*stk)-- = (unsigned long)0x08; // CS
    *(*stk)-- = (unsigned long)task; // eip
    // pushf
    *(*stk)-- = (unsigned long)0x0202; // flag registers
    // pusha
    *(*stk)-- = (unsigned long)0xAAAAAAAA; // eax
    *(*stk)-- = (unsigned long)0xCCCCCCCC; // ecx
    *(*stk)-- = (unsigned long)0xDDDDDDDD; // edx
    *(*stk)-- = (unsigned long)0BBBBBBB; // ebx
    *(*stk)-- = (unsigned long)0x44444444; // esp
    *(*stk)-- = (unsigned long)0x55555555; // ebp
    *(*stk)-- = (unsigned long)0x66666666; // esi
    **stk      = (unsigned long)0x77777777; // edi
}

```

此外新增模块还有用户层的测试函数, 由于其与任务管理相关不大, 且较为简单, 在具体运行时才有相关, 故不在此处涉及

- 代码组织及其实现
 - 目录组织





代码按模块主要分为四部分：

1. multibootheader

此模块提供multibootHeader段的代码，使得boot loader成功将操作系统载入内存

2. myOS

此模块提供操作系统的代码，包括

start32.S 初始化C程序的运行环境，设定IDT的汇编级程序

dev 提供硬件层UART, VGA的使用封装接口, i8253, i8259A的初始化接口

i386 提供基于i386架构的底层硬件IO的接口, 中断处理程序

include C程序的头文件

lib C程序的库

printk myPrintk, vsprintf函数的实现

kernel 时钟显示, 时钟定时模块的实现, 内存管理模块的实现, 任务管理模块的实现

osStart.c 初始化系统并进入用户程序

3. userApp

此模块存放用户程序，本次实验包含

main.c, shell.c, memTestCase.c, schedulerTestCase.c, 提供shell终端服务, 以及内存管理测试, 任务调度测试

4. output

此模块存放含操作系统及用户程序在内的所有程序根据Makefile, ld文件编译链接后的生成的可执行文件，结构与myOS类似，不再赘述

o Makefile组织

```
.
├── MULTI_BOOT_HEADER
```

```

|   └─ output/multibootheader/multibootHeader.o
└─ OS_OBJS
    └─ MYOS_OBJS
        │   └─ output/myOS/start32.o
        │   └─ output/myOS/osStart.o
        │   └─ DEV_OBJS
        │       │   └─ output/myOS/dev/uart.o
        │       │   └─ output/myOS/dev/vga.o
        │       │   └─ output/myOS/dev/i8259A.o
        │       │   └─ output/myOS/dev/i8253.o
        │       └─ I386_OBJS
        │           │   └─ output/myOS/i386/io.o
        │           │   └─ output/myOS/i386/irqs.o
        │           │   └─ output/myOS/i386/ctx_sw.o
        │           │   └─ output/myOS/i386/irq.o
        │           └─ PRINTK_OBJS
        │               └─ output/myOS/printk/myPrintk.o
        └─ LIB_OBJS
            └─ output/myOS/lib/string.o
        └─ KERNEL_OBJS
            │   └─ output/myOS/kernel/tick.o
            │   └─ output/myOS/kernel/wallClock.o
            │   └─ TASK_OBJS
            │       │   └─ output/myOS/kernel/task/scheduler.o
            │       │   └─ output/myOS/kernel/task/task.o
            │       │   └─ output/myOS/kernel/task/taskArrv.o
            │       │   └─ output/myOS/kernel/task/taskQueue.o
            │       │   └─ output/myOS/kernel/task/taskPara.o
            │       └─ TIME_OBJS
            │           │   └─ output/myOS/kernel/time/tick.o
            │           │   └─ output/myOS/kernel/time/timer.o
            │           │   └─ output/myOS/kernel/time/wallClock.o
            │           │   └─ output/myOS/kernel/time/hook.o
            │           └─ MEM_OBJS
            │               │   └─ output/myOS/kernel/mem/pMemInit.o
            │               │   └─ output/myOS/kernel/mem/dPartition.o
            │               │   └─ output/myOS/kernel/mem/eFPartition.o
            │               │   └─ output/myOS/kernel/mem/malloc.o
            └─ USER_APP_OBJS
                │   └─ output/userApp/main.o
                │   └─ output/userApp/shell.o
                │   └─ output/userApp/schedulerTestCase.o
                └─ output/userApp/memTestCase.o

```

- 代码布局说明

借助C程序可查看各段的起止位置

```

#include "vga.h"
#include "myPrintk.h"
extern unsigned long __multiboot_start;
extern unsigned long __multiboot_end;
extern unsigned long __text_start;
extern unsigned long __text_end;
extern unsigned long __data_start;

```

```

extern unsigned long __data_end;
extern unsigned long __bss_start;
extern unsigned long __bss_end;
/* 此文件无需修改 */

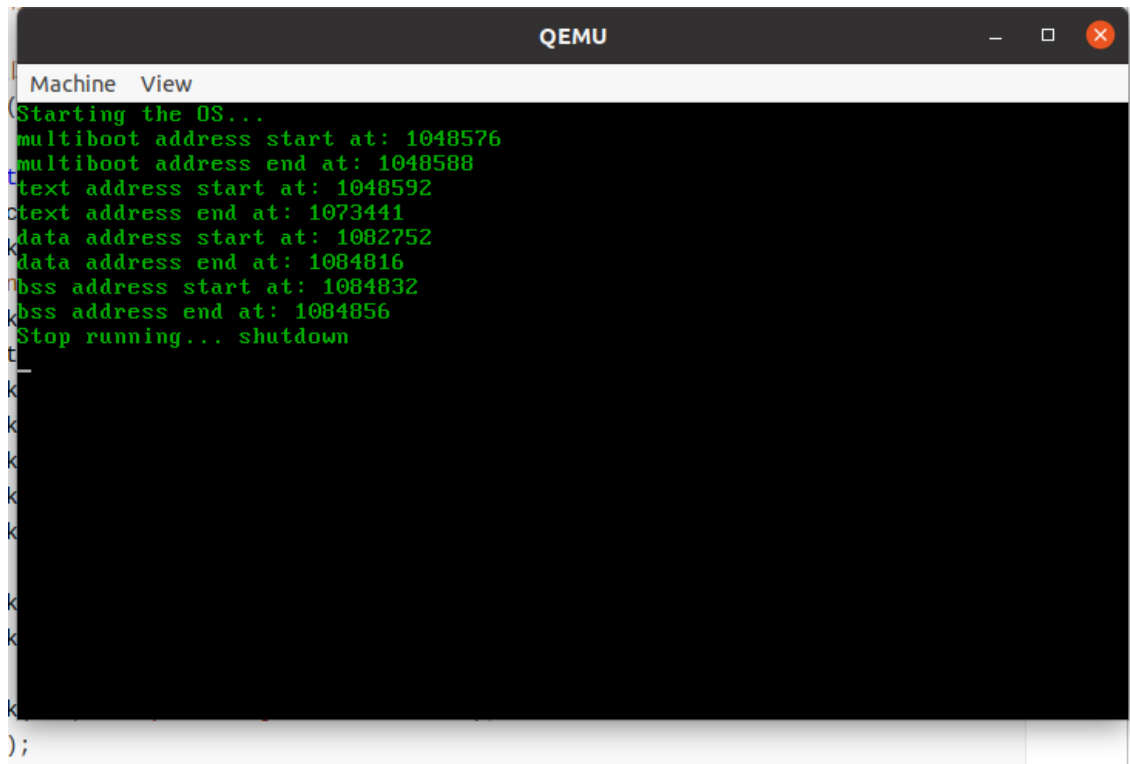
// 用户程序入口
void myMain(void);

void osStart(void) {
    clear_screen();
    myPrintk(0x2, "Starting the OS...\n");
    //myMain();
    myPrintk(0x2, "multiboot address start at: %d\n", (unsigned
long)&__multiboot_start);
    myPrintk(0x2, "multiboot address end at: %d\n", (unsigned
long)&__multiboot_end);
    myPrintk(0x2, "text address start at: %d\n", (unsigned long)&__text_start);
    myPrintk(0x2, "text address end at: %d\n", (unsigned long)&__text_end);
    myPrintk(0x2, "data address start at: %d\n", (unsigned long)&__data_start);
    myPrintk(0x2, "data address end at: %d\n", (unsigned long)&__data_end);

    myPrintk(0x2, "bss address start at: %d\n", (unsigned long)&__bss_start);
    myPrintk(0x2, "bss address end at: %d\n", (unsigned long)&__bss_end);

    myPrintk(0x2, "Stop running... shutdown\n");
    while(1);
}

```



The screenshot shows a QEMU window titled "QEMU" with a "Machine View" tab. The terminal output is as follows:

```

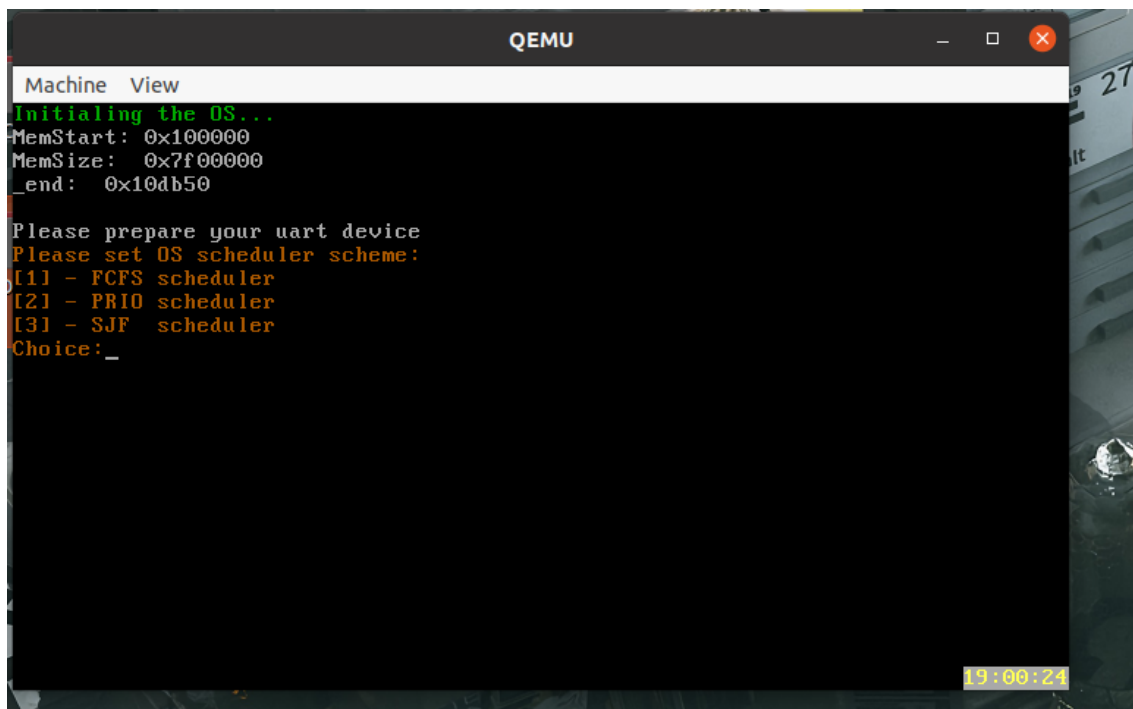
Starting the OS...
multiboot address start at: 1048576
multiboot address end at: 1048588
text address start at: 1048592
text address end at: 1073441
data address start at: 1082752
data address end at: 1084816
bss address start at: 1084832
bss address end at: 1084856
Stop running... shutdown

```

Section	Offset(Base = 0)
.multiboot_header	0x00100000~0x0010000c
.text	0x00100010~0x00106121
.data	0x00108580~0x00108d90
.bss	0x00108da0~0x00108db8

- 运行测试

本次实验中由于首先创建了init(tid为1)与idle(tid为0), 因此后续任务分配的tid由2开始, 在init执行结束并销毁后即测试任务运行序列



- o FCFS

A screenshot of a QEMU terminal window. The title bar says 'QEMU'. Below it is a 'Machine View' header. The terminal output shows the initialization of user tasks and the execution of five tasks (Tid 1 to Tid 5) in a First-Come-First-Served (FCFS) manner. The timer starts at 0. Task 1 (Tid 1) arrives at time 0 and ends at time 1. Task 5 (Tid 5) arrives at time 0 and ends at time 3. Task 2 (Tid 2) arrives at time 3 and ends at time 8. Task 3 (Tid 3) arrives at time 10 and ends at time 16. Task 4 (Tid 4) arrives at time 13 and ends at time 20. The output is color-coded: green for arrival, orange for task end, and purple for task start. A timestamp '19:01:53' is visible in the bottom right corner.

```
Machine View

Init Task:myMain() ----- Initing user tasks
*****
Now timer is 0,Tid 5 arrive
task end! tid :1
Tasknow id is 5
myTskFCFS5 Begin,the timer is 0
Now timer is 3,Tid 2 arrive
myTskFCFS5 End,the timer is 3
task end! tid :5
Tasknow id is 2
myTskFCFS2 Begin,the timer is 3
myTskFCFS2 End,the timer is 8
task end! tid :2
Now timer is 10,Tid 3 arrive
Tasknow id is 3
myTskFCFS3 Begin,the timer is 10
Now timer is 13,Tid 4 arrive
myTskFCFS3 End,the timer is 16
task end! tid :3
Tasknow id is 4
myTskFCFS4 Begin,the timer is 16
myTskFCFS4 End,the timer is 20
task end! tid :4

19:01:53
```

```
void myTskFCFS2(void) {
    myPrintf(0x07,"myTskFCFS2 Begin,the timer is %d\n",get_time());
    task_execute(4);
    myPrintf(0x07,"myTskFCFS2 End,the timer is %d\n",get_time());
    tskEnd();
}

void myTskFCFS3(void) {
    myPrintf(0x07,"myTskFCFS3 Begin,the timer is %d\n",get_time());
    task_execute(5);
    myPrintf(0x07,"myTskFCFS3 End,the timer is %d\n",get_time());
    tskEnd();
}

void myTskFCFS4(void) {
    myPrintf(0x07,"myTskFCFS4 Begin,the timer is %d\n",get_time());
    task_execute(3);
    myPrintf(0x07,"myTskFCFS4 End,the timer is %d\n",get_time());
    tskEnd();
}

void myTskFCFS5(void) {
    myPrintf(0x07,"myTskFCFS5 Begin,the timer is %d\n",get_time());
    task_execute(2);
    myPrintf(0x07,"myTskFCFS5 End,the timer is %d\n",get_time());
    tskEnd();
}

void initFCFSCases(void) {
    int newTskTid2 = createTsk(myTskFCFS2); // its tid will be 2
    setTskPara(Arrv_Time, 3, taskctl.tcbPool[newTskTid2].para);
    setTskPara(EXEC_Time, 4, taskctl.tcbPool[newTskTid2].para);

    int newTskTid3 = createTsk(myTskFCFS3); // its tid will be 3
    setTskPara(Arrv_Time, 10, taskctl.tcbPool[newTskTid3].para);
}
```

```

setTskPara(EXEC_TIME, 5, taskctl.tcbPool[newTskTid3].para);

int newTskTid4 = createTsk(myTskFCFS4); // its tid will be 4
setTskPara(ARRV_TIME, 13, taskctl.tcbPool[newTskTid4].para);
setTskPara(EXEC_TIME, 3, taskctl.tcbPool[newTskTid4].para);

int newTskTid5 = createTsk(myTskFCFS5); // its tid will be 5
setTskPara(ARRV_TIME, 0, taskctl.tcbPool[newTskTid5].para);
setTskPara(EXEC_TIME, 2, taskctl.tcbPool[newTskTid5].para);

enableTask(newTskTid2);

enableTask(newTskTid3);

enableTask(newTskTid4);

enableTask(newTskTid5);
}

```

任务	到达时间	运行时间	执行时间段
tid2	3	4	3-8
tid3	10	5	10-16
tid4	13	3	16-20
tid5	0	2	0-3

由上表可知, 执行序列符合FCFS

o PRIO

```

Tasknow id is 3
myTskPrio3 Begin,the timer is 1
Now timer is 1,Tid 4 arrive
Tasknow id is 4
myTskPrio4 Begin,the timer is 1
Now timer is 1,Tid 5 arrive
Now timer is 4,Tid 6 arrive
myTskPrio4 End,the timer is 5
task end! tid :4
Tasknow id is 3
myTskPrio3 End,the timer is 9
task end! tid :3
Tasknow id is 2
myTskPrio2 End,the timer is 13
task end! tid :2
Tasknow id is 5
myTskPrio5 Begin,the timer is 13
myTskPrio5 End,the timer is 17
task end! tid :5
Tasknow id is 6
myTskPrio4 Begin,the timer is 17
myTskPrio4 End,the timer is 21
task end! tid :6

```

19:11:46

```

press any key to start!
Starting the OS...
START MULTITASKING.....
Tasknow id is 1
*****
Init Task:myMain() ----- Initiing user tasks
*****
Now timer is 0,Tid 2 arrive
task end! tid :1
Tasknow id is 2
myTskPrio2 Begin,the timer is 0
Now timer is 1,Tid 3 arrive
Tasknow id is 3
myTskPrio3 Begin,the timer is 1
Now timer is 1,Tid 4 arrive
Tasknow id is 4
myTskPrio4 Begin,the timer is 1
Now timer is 1,Tid 5 arrive
Now timer is 4,Tid 6 arrive
myTskPrio4 End,the timer is 5
task end! tid :4
Tasknow id is 3
myTskPrio3 End,the timer is 9
task end! tid :3
Tasknow id is 2
myTskPrio2 End,the timer is 13
task end! tid :2
Tasknow id is 5
myTskPrio5 Begin,the timer is 13
myTskPrio5 End,the timer is 17
task end! tid :5
Tasknow id is 6
myTskPrio4 Begin,the timer is 17
myTskPrio4 End,the timer is 21
task end! tid :6

```

任务	到达时间	运行时间	优先级	执行时间段
tid2	0	4	3	0-1,9-13
tid3	1	3	1	1-1,5-9
tid4	1	3	0	1-5
tid5	1	3	4	13-17
tid6	4	3	4	17-21

由上表可知,该优先级调度算法是抢占式,符合优先级高的先调度(优先级越小,优先级越高),故调度序列符合要求

o SJF

```
Tasknow id is 2
myTskSJF2 Begin,the timer is 0
myTskSJF2 End,the timer is 3
task end! tid :2
Tasknow id is 4
myTskSJF4 Begin,the timer is 3
myTskSJF4 End,the timer is 8
task end! tid :4
Tasknow id is 3
myTskSJF3 Begin,the timer is 8
Now timer is 10,Tid 5 arrive
Now timer is 12,Tid 6 arrive
myTskSJF3 End,the timer is 14
task end! tid :3
Tasknow id is 5
myTskSJF5 Begin,the timer is 14
myTskSJF5 End,the timer is 18
task end! tid :5
Tasknow id is 6
myTskSJF6 Begin,the timer is 18
myTskSJF6 End,the timer is 22
task end! tid :6
Tasknow id is 0
-
19:00:49
```

任务	到达时间	运行时间	执行时间段
tid2	0	2	0-3
tid3	0	5	8-14
tid4	0	4	3-8
tid5	10	3	14-18
tid6	12	3	18-22

由上表可知,该调度算法是非抢占式,且执行时间短的先调度,故调度序列符合要求

备注:表格中的执行时间段为程序从开始执行到执行结束的时间段,不包含时间段的左边界,故执行时间与运行时间一致

【问题与解决】

1. 对任务切换理解不清

本次实验的主要问题在于一开始对进程上下文的理解不深, 造成困惑, 实际上本次实验并未显示的进行任务调度, 而是借助栈的切换实现运行不同的函数体以实现多任务调度

2. 任务结束后未显式调用任务结束, 导致任务无法返回调度程序

改正后可正常运行

【附录】

src1为lab5的完成代码

src2为lab6的完成代码