实验六，调度算法

李卓 pb19000064

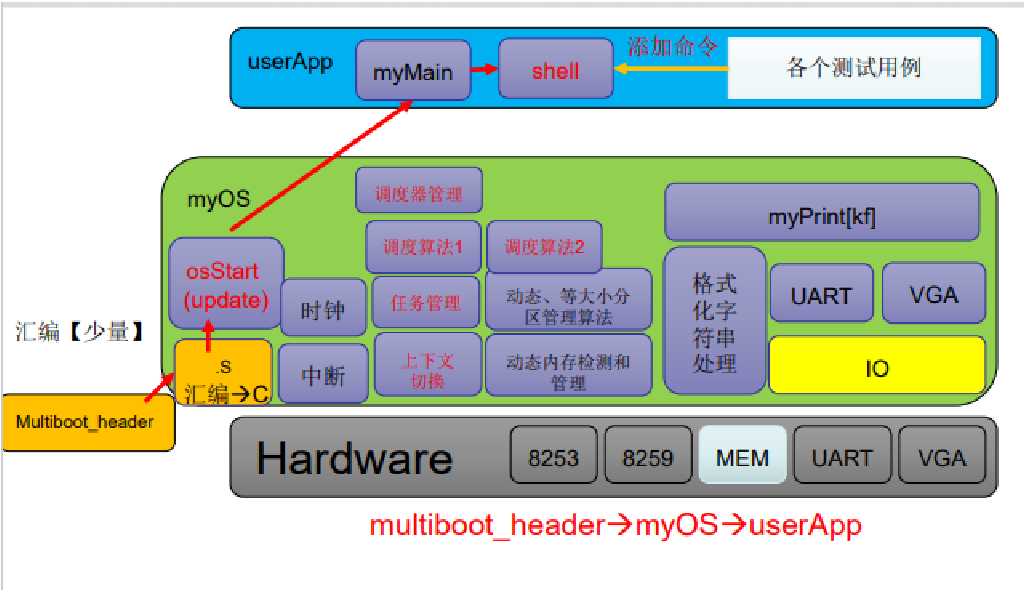
实验目的

1. 实现调度算法，至少 2 种（不含 FCFS）
2. 实现支持调度算法的任务管理器
3. 根据调度算法需要修改，任务数据结构，任务创建/销毁，调度器

实验内容

1. 实现任务随时钟动态化到达
2. 调度器和任务参数采用统一接口
3. 完成一种抢占式调度算法
4. 完成另外两种任意调度算法

实验框架



内核：上下文切换、任务管理和调度

用户：新功能测试

被测功能：任务创建、所实现的调度算法

自测：userApp

实验流程

1. 在 multiboot_header 中完成系统的启动。

2. 在 start32.S 中做好准备，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，调用 myMain， 进入 userApp 部分。

4. 运行 myMain 中的代码，进行时钟设置，shell 初始化，内存测试初始化等操作，启动 shell。

5. 进入 shell 程序，等待命令的输入

Multiboot_header 为进入 C 程序准备好上下文 初始化操作系统各个模块调用 userApp 入口 myMain（自测）+shell

实验原理

Tcb 结构 以及 tcb 池

```
typedef struct myTCB {
    struct dLink_node thisNode;

    int tcbIndex;
    tskPara para;
    unsigned long state;

    struct myTCB * next;
    unsigned long* stkTop;
    unsigned long stack[STACK_SIZE];
} myTCB;

#define TASK_NUM (2 + USER_TASK_NUM)
myTCB tcbPool[TASK_NUM];
```

Tcb 参数及其操作

```
tskPara defaultTskPara = {
    .priority = MAX_PRIORITY_NUM,
    .exeTime = MAX_EXETIME,
    .arrTime = 0,
    .schedPolicy = SCHED_UNDEF};

void copyTskPara(myTCB *task, tskPara *para){ ... }

void initTskPara(tskPara *buffer){ ... }

void setTskPara(unsigned int option, unsigned int value, t
```

Tsk 使用链表存储

Tsk 创建:

createTsk()实现 TCB 分配, 对调度参数和栈初始化, 对下一空闲 TCB 进行修改, 若 此时为到达时间, 直接调用 tskStart()启动任务. 否则调用 tskPreStart()函数对 tsk 放置在合适位置。

```
int createTsk(void (*tskBody)(void), tskPara *para)
{
    myTCB *allocated = firstFreeTsk;
    if (firstFreeTsk == NULL)
        return -1;
    firstFreeTsk = allocated->next;
    allocated->next = NULL;
    copyTskPara(allocated, para);
    stack_init(&(allocated->stkTop), tskBody);
    createTsk_hook(allocated);
    if (allocated->para.arrTime == 0)
        tskStart(allocated);
    else
        tskPreStart(allocated);
    return allocated->tcbIndex;
}
```

Tsk 销毁

destroyTsk()实现 TCB 回收, 修改 TCB 链表, 同时调 度新任务。

```
void destroyTsk(int tskIndex)
{
    tcbPool[tskIndex].next = firstFreeTsk;
    firstFreeTsk = &tcbPool[tskIndex];
    schedule();
}
```

调度算法:

Scheduler 结构

```c
struct scheduler {
    unsigned int type;
    myTCB* (*nextTsk_func)(void);
    void (*enqueueTsk_func)(myTCB *tsk);
    void (*dequeueTsk_func)(myTCB *tsk);
    void (*schedulerInit_func)(void);
    void (*createTsk_hook)(myTCB* created);
    void (*tick_hook)(void);
};
```

实现了统一的调度接口

```c
extern myTCB *curTsk;

extern void context_switch(myTCB *prevTsk, myTCB *nextTsk);

struct scheduler *sysScheduler = &scheduler_FCFS;

unsigned int getSysScheduler(void){ ... }

void setSysScheduler(unsigned int method){ ... }

myTCB *nextTsk(void){ ... }

void enqueueTsk(myTCB *tsk){ ... }

void dequeueTsk(myTCB *tsk){ ... }

void createTsk_hook(myTCB *created){ ... }

extern void scheduler_hook_main(void);
void schedulerInit(){ ... }
void schedule(void){ ... }
```

利用 hook 机制配置相应算法

```c
struct scheduler scheduler_FCFS = {
    .type = SCHEDULER_FCFS,
    .nextTsk_func = nextTsk_FCFS,
    .enqueueTsk_func = EnqueueTsk_FCFS,
    .dequeueTsk_func = DequeueTsk_FCFS,
    .schedulerInit_func = schedulerInit_FCFS,
    .createTsk_hook = NULL,
    .tick_hook = NULL
};
```

实现了 Prio, FCFS, SJF 算法

Prio 算法:

```c
void EnqueueTsk_PRIO(myTCB *tsk)
{
    myTCB *point;
    point = PRIORdyTCB;
    if (point == NULL)
        dLinkInsertBefore((dLinkedList *)point, (dLink_node *)point, (dLink_node *)tsk);
    else
    {
        while (tsk->para.priority > point->para.priority && point->next != 0)
            point = point->next;
        if (tsk->para.priority >= point->para.priority)
            dLinkInsertAfter((dLinkedList *)point, (dLink_node *)point, (dLink_node *)tsk);
        else
            dLinkInsertBefore((dLinkedList *)point, (dLink_node *)point, (dLink_node *)tsk);
    }
}
```

FCFS, SJF 算法与 PRIO 本质相同，都是将 tsk 序列排序，只不过 FCFS 优先级是到达时间, SJF 优先级是剩余运行时间.

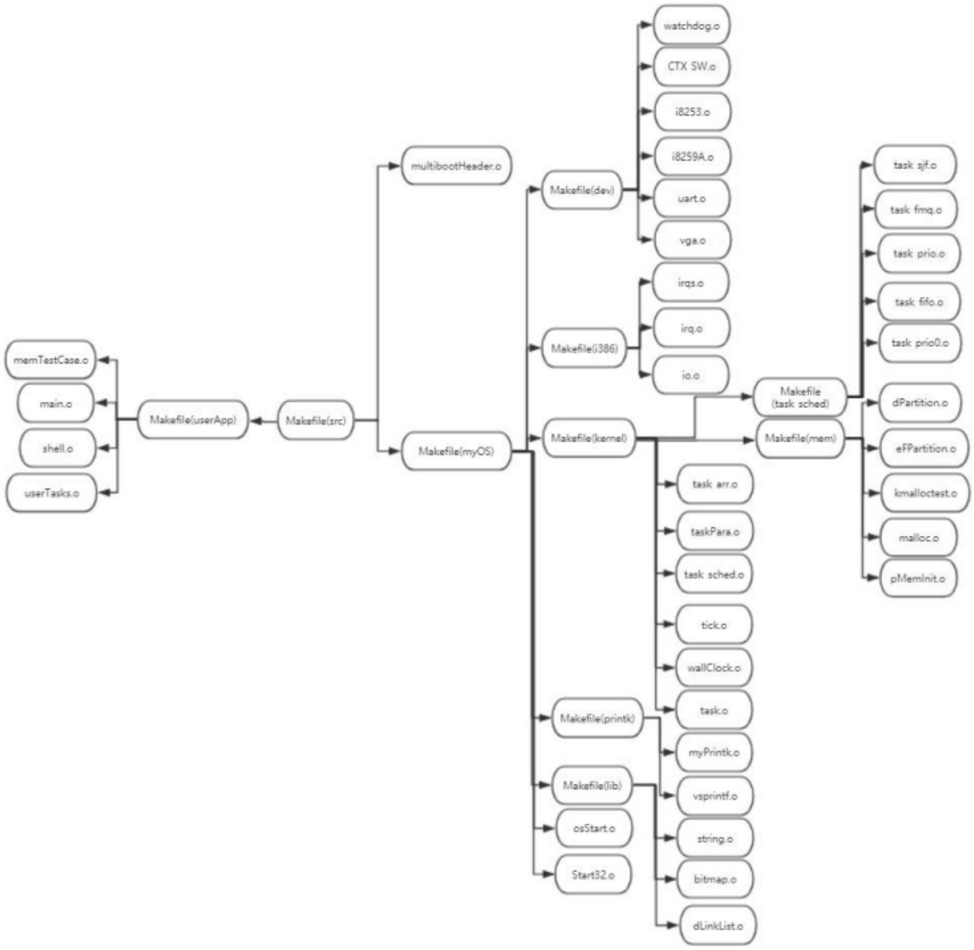
文件目录组织

```
├── Makefile
├── multibootheader
│   └── multibootHeader.S
├── myOS
│   ├── dev
│   │   ├── i8253.c
│   │   ├── i8259A.c
│   │   ├── Makefile
│   │   ├── uart.c
│   │   ├── vga.c
│   │   └── watchdog.c
│   ├── i386
│   │   ├── CTX_SW.S
│   │   ├── io.c
│   │   ├── irq.S
│   │   ├── irqs.c
│   │   └── Makefile
│   ├── include
│   │   ├── dLinkList.h
│   │   ├── i8253.h
│   │   ├── i8259.h
│   │   ├── io.h
│   │   ├── irq.h
│   │   ├── kmalloc.h
│   │   ├── malloc.h
│   │   ├── mem.h
│   │   ├── myPrintk.h
│   │   ├── string.h
│   │   ├── task_arr.h
│   │   ├── task.h
│   │   ├── taskPara.h
│   │   ├── task_sched.h
│   │   ├── tcb.h
│   │   ├── time.h
│   │   ├── uart.h
│   │   ├── vga.h
│   │   ├── vsprintf.h
│   │   ├── wallClock.h
│   │   └── watchdog.h

│   │   └── watchdog.h
│   ├── kernel
│   │   ├── Makefile
│   │   ├── mem
│   │   │   ├── dPartition.c
│   │   │   ├── eFPartition.c
│   │   │   ├── Makefile
│   │   │   ├── malloc.c
│   │   │   └── pMemInit.c
│   │   ├── task_arr.c
│   │   ├── task.c
│   │   ├── taskPara.c
│   │   ├── task_sched
│   │   │   ├── Makefile
│   │   │   ├── task_fifo.c
│   │   │   ├── task_prio.c
│   │   │   └── task_sjf.c
│   │   ├── task_sched.c
│   │   ├── tick.c
│   │   └── wallClock.c
│   ├── lib
│   │   ├── dLinkList.c
│   │   ├── Makefile
│   │   └── string.c
│   ├── Makefile
│   ├── myOS.ld
│   ├── osStart.c
│   ├── printk
│   │   ├── Makefile
│   │   ├── myPrintk.c
│   │   ├── types.h
│   │   └── vsprintf.c
│   ├── start32.S
│   └── userInterface.h
├── output

├── source2img.sh
├── tests
│   ├── test0_fcfs
│   │   ├── main.c
│   │   ├── Makefile
│   │   ├── memTestCase.c
│   │   ├── memTestCase.h
│   │   ├── shell.c
│   │   ├── shell.h
│   │   └── userApp.h
│   ├── test1_prio
│   │   ├── main.c
│   │   ├── Makefile
│   │   ├── memTestCase.c
│   │   ├── memTestCase.h
│   │   ├── myOS.elf
│   │   ├── shell.c
│   │   ├── shell.h
│   │   └── userApp.h
│   └── test2_sjf
│       ├── main.c
│       ├── Makefile
│       ├── memTestCase.c
│       ├── memTestCase.h
│       ├── myOS.elf
│       ├── shell.c
│       ├── shell.h
│       └── userApp.h
└── userApp
    ├── main.c
    ├── Makefile
    ├── memTestCase.c
    ├── memTestCase.h
    ├── myOS.elf
    ├── shell.c
    ├── shell.h
    └── userApp.h
```

makefile 组织



地址空间布局

Section	Offset (Base = 1M)	align
.multiboot_header	0	8
.text(代码段)	16	8
.data(数据段)	16+.text section	16
.bss	当前	16
堆栈(动态内存空间)	当前	

编译过程说明

默认方式，链接生成 myOS.elf 文件

chmod 777 source2run.sh

./source2run.sh test0_fcfs

./source2run.sh test1_prio

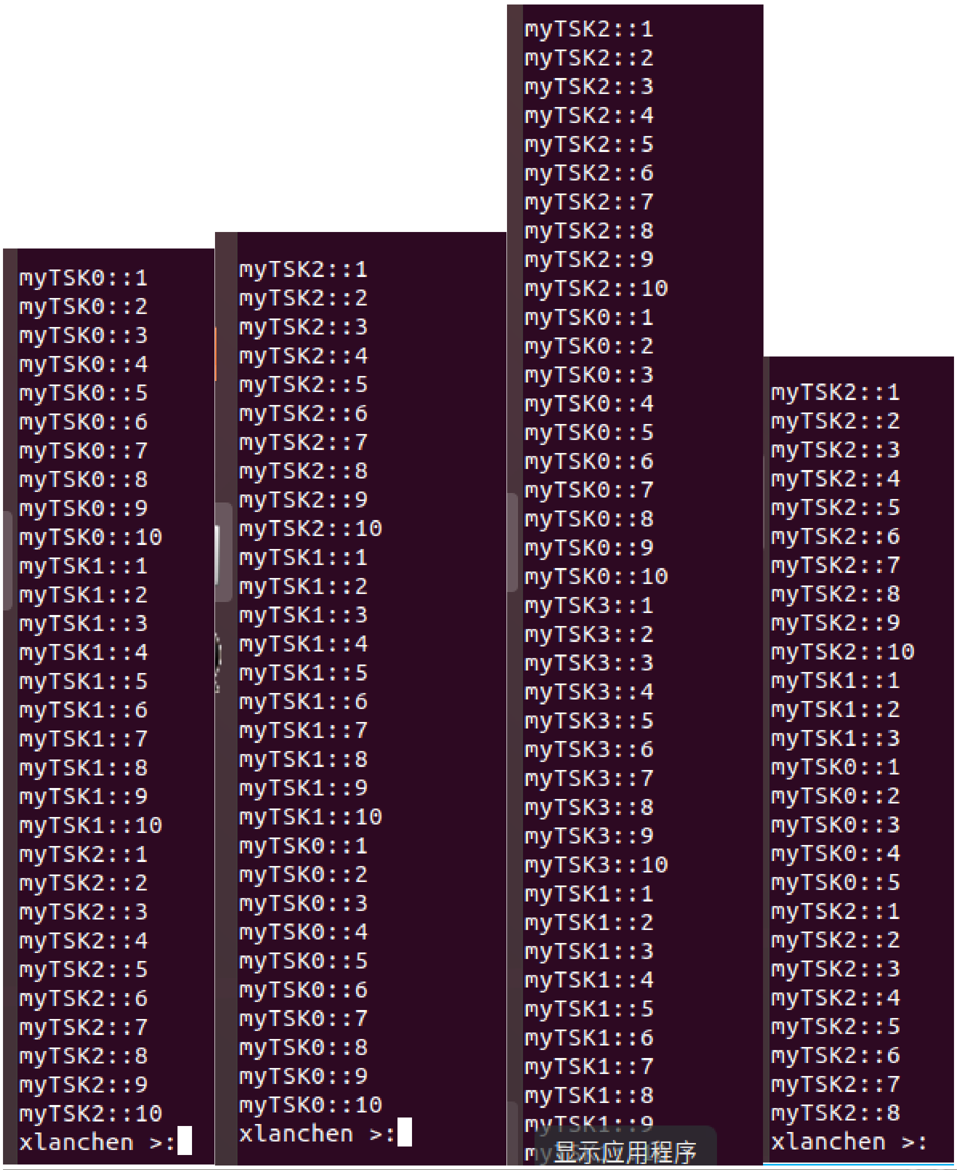
./source2run.sh test2_fjs

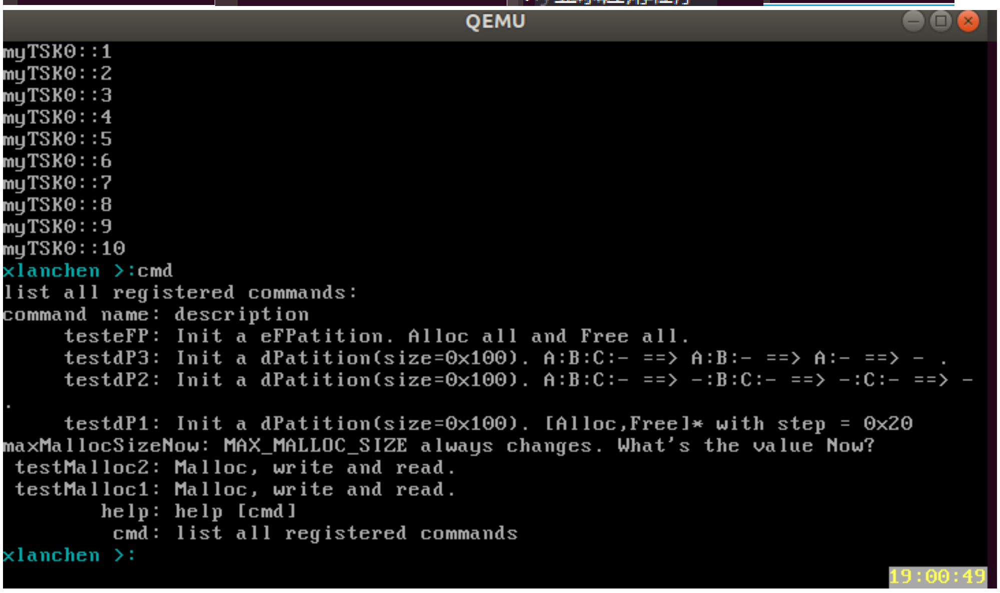
sudo screen /dev/pts/1

运行结果

```
lz@ubuntu:/mnt/hgfs/workspace/lab56/os-lab6$ ./source2img.sh test0_fcfs
test0_fcfs ====
===========================make allclean=======================================
============编译链接test0_fcfs并制作成二进制映像============
rm -rf output
*********make test0_fcfs *********
ld -n -T myOS/myOS.ld output/multibootheader/multibootHeader.o output/myOS/start
put/myOS/i386/io.o output/myOS/i386/irq.o output/myOS/i386/irqs.o output/myOS/i3
tput/myOS/kernel/wallClock.o output/myOS/kernel/task.o output/myOS/kernel/task_a
dPartition.o output/myOS/kernel/mem/eFPartition.o output/myOS/kernel/mem/malloc.
.o  output/userApp/main.o output/userApp/shell.o output/userApp/memTestCase.o -o
make succeed
```

```
lz@ubuntu:/mnt/hgfs/workspace/lab56/os-lab6$ ./source2img.sh test2_sjf
test2_sjf ====
===========================make allclean=====================================
============编译链接test2_sjf并制作成二进制映像============
rm -rf output
*********make test2_sjf *********
ld -n -T myOS/myOS.ld output/multibootheader/multibootHeader.o output/myOS/start32.o output/myOS/
put/myOS/i386/io.o output/myOS/i386/irq.o output/myOS/i386/irqs.o output/myOS/i386/CTX_SW.o outpu
tput/myOS/kernel/wallClock.o output/myOS/kernel/task.o output/myOS/kernel/task_arr.o output/myOS/
dPartition.o output/myOS/kernel/mem/eFPartition.o output/myOS/kernel/mem/malloc.o output/myOS/ker
o0.o  output/userApp/main.o output/userApp/shell.o output/userApp/memTestCase.o -o output/myOS.el
make succeed
```


myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK0::7
myTSK0::8
myTSK0::9
myTSK0::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK1::4
myTSK1::5
myTSK1::6
myTSK1::7
myTSK1::8
myTSK1::9
myTSK1::10
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
xlanchen >:
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK1::4
myTSK1::5
myTSK1::6
myTSK1::7
myTSK1::8
myTSK1::9
myTSK1::10
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK0::7
myTSK0::8
myTSK0::9
myTSK0::10
xlanchen >:
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK0::7
myTSK0::8
myTSK0::9
myTSK0::10
myTSK3::1
myTSK3::2
myTSK3::3
myTSK3::4
myTSK3::5
myTSK3::6
myTSK3::7
myTSK3::8
myTSK3::9
myTSK3::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK1::4
myTSK1::5
myTSK1::6
myTSK1::7
myTSK1::8
myTSK1::9
显示应用程序
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
xlanchen >:


QEMU
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK0::7
myTSK0::8
myTSK0::9
myTSK0::10
xlanchen >:cmd
list all registered commands:
command name: description
testeFP: Init a eFPatition. Alloc all and Free all.
testdP3: Init a dPatition(size=0x100). A:B:C:- ==> A:B:- ==> A:- ==> - .
testdP2: Init a dPatition(size=0x100). A:B:C:- ==> -:B:C:- ==> -:C:- ==> - .
testdP1: Init a dPatition(size=0x100). [Alloc,Free]* with step = 0x20
maxMallocSizeNow: MAX_MALLOC_SIZE always changes. What's the value Now?
testMalloc2: Malloc, write and read.
testMalloc1: Malloc, write and read.
help: help [cmd]
cmd: list all registered commands
xlanchen >:
19:00:49

运行结果解释

Fcfs:

```
setTskPara(ARRTIME,0,&tskParas[0]);
createTsk(myTSK0,&tskParas[0]);

setTskPara(ARRTIME,5,&tskParas[1]);
createTsk(myTSK1,&tskParas[1]);

setTskPara(ARRTIME,10,&tskParas[2]);
createTsk(myTSK2,&tskParas[2]);
```

可以明显看到三个 task 按照到来时间依次执行

```
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK0::7
myTSK0::8
myTSK0::9
myTSK0::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK1::4
myTSK1::5
myTSK1::6
myTSK1::7
myTSK1::8
myTSK1::9
myTSK1::10
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
xlanchen >:
```

```
setTskPara(ARRTIME,10,&tskParas[0]);
createTsk(myTSK0,&tskParas[0]);

setTskPara(ARRTIME,5,&tskParas[1]);
createTsk(myTSK1,&tskParas[1]);

setTskPara(ARRTIME,0,&tskParas[2]);
createTsk(myTSK2,&tskParas[2]);
```

修改到来时间反转，可以看到倒序执行

```
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK1::4
myTSK1::5
myTSK1::6
myTSK1::7
myTSK1::8
myTSK1::9
myTSK1::10
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK0::7
myTSK0::8
myTSK0::9
myTSK0::10
xlanchen >:
```

非抢占 prio:

```c
setTskPara(ARRTIME, 50, &tskParas[0]);
setTskPara(PRIORITY, 1, &tskParas[0]);
createTsk(myTSK0, &tskParas[0]);

setTskPara(ARRTIME, 100, &tskParas[1]);
setTskPara(PRIORITY, 1, &tskParas[1]);
createTsk(myTSK1, &tskParas[1]);

setTskPara(ARRTIME, 0, &tskParas[2]);
setTskPara(PRIORITY, 2, &tskParas[2]);
createTsk(myTSK2, &tskParas[2]);
setTskPara(ARRTIME, 100, &tskParas[3]);
setTskPara(PRIORITY, 0, &tskParas[3]);
createTsk(myTSK3, &tskParas[3]);
```

Task2 task3 同时到达，但是 task2 优先级高，先执行 task2. 然后 task0 到达，然后 task2 结束，优先选择后来的 task0，直到 task0 结束, task3 才得以执行

```
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
myTSK0::7
myTSK0::8
myTSK0::9
myTSK0::10
myTSK3::1
myTSK3::2
myTSK3::3
myTSK3::4
myTSK3::5
myTSK3::6
myTSK3::7
myTSK3::8
myTSK3::9
myTSK3::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK1::4
myTSK1::5
myTSK1::6
myTSK1::7
myTSK1::8
myTSK1::9
my 显示应用程序
```

抢占式 sjf:

```
setTskPara(ARRTIME, 100, &tskParas[0]);
setTskPara(EXETIME, 5, &tskParas[0]);
createTsk(myTSK0, &tskParas[0]);

setTskPara(ARRTIME, 100, &tskParas[1]);
setTskPara(EXETIME, 3, &tskParas[1]);
createTsk(myTSK1, &tskParas[1]);

setTskPara(ARRTIME, 0, &tskParas[2]);
setTskPara(EXETIME, 18, &tskParas[2]);
createTsk(myTSK2, &tskParas[2]);
```

Task2 最先到达并开始执行，执行到第 100tick 时, task0 和 task1 到达，算法开始调度．此时 task2 只执行到第 10 步还剩余 8 部分，由于算法为抢占式，运行时间更短的 task1 开始执行，然后是 task2，都结束最后才轮到 task2 的剩余部分

```
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
myTSK1::1
myTSK1::2
myTSK1::3
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
xlanchen >:
```

```c
setTskPara(ARRTIME, 120, &tskParas[0]);
setTskPara(EXETIME, 6, &tskParas[0]);
createTsk(myTSK0, &tskParas[0]);

setTskPara(ARRTIME, 120, &tskParas[1]);
setTskPara(EXETIME, 2, &tskParas[1]);
createTsk(myTSK1, &tskParas[1]);

setTskPara(ARRTIME, 0, &tskParas[2]);
setTskPara(EXETIME, 10, &tskParas[2]);
createTsk(myTSK2, &tskParas[2]);
```

在 120ticks 时, task0 和 task1 同时到达，选择了时间更短的 task1

```
********INIT START

********INIT END

myTSK2::1
myTSK2::2
myTSK2::3
myTSK2::4
myTSK2::5
myTSK2::6
myTSK2::7
myTSK2::8
myTSK2::9
myTSK2::10
..........IDLE...............0.
myTSK1::1
myTSK1::2
myTSK0::1
myTSK0::2
myTSK0::3
myTSK0::4
myTSK0::5
myTSK0::6
xlanchen >:
```

实验中遇到的问题

1. 没理清文件结构，对全局变量重定义

2. 使用指针前,忘记判断是否为空指针