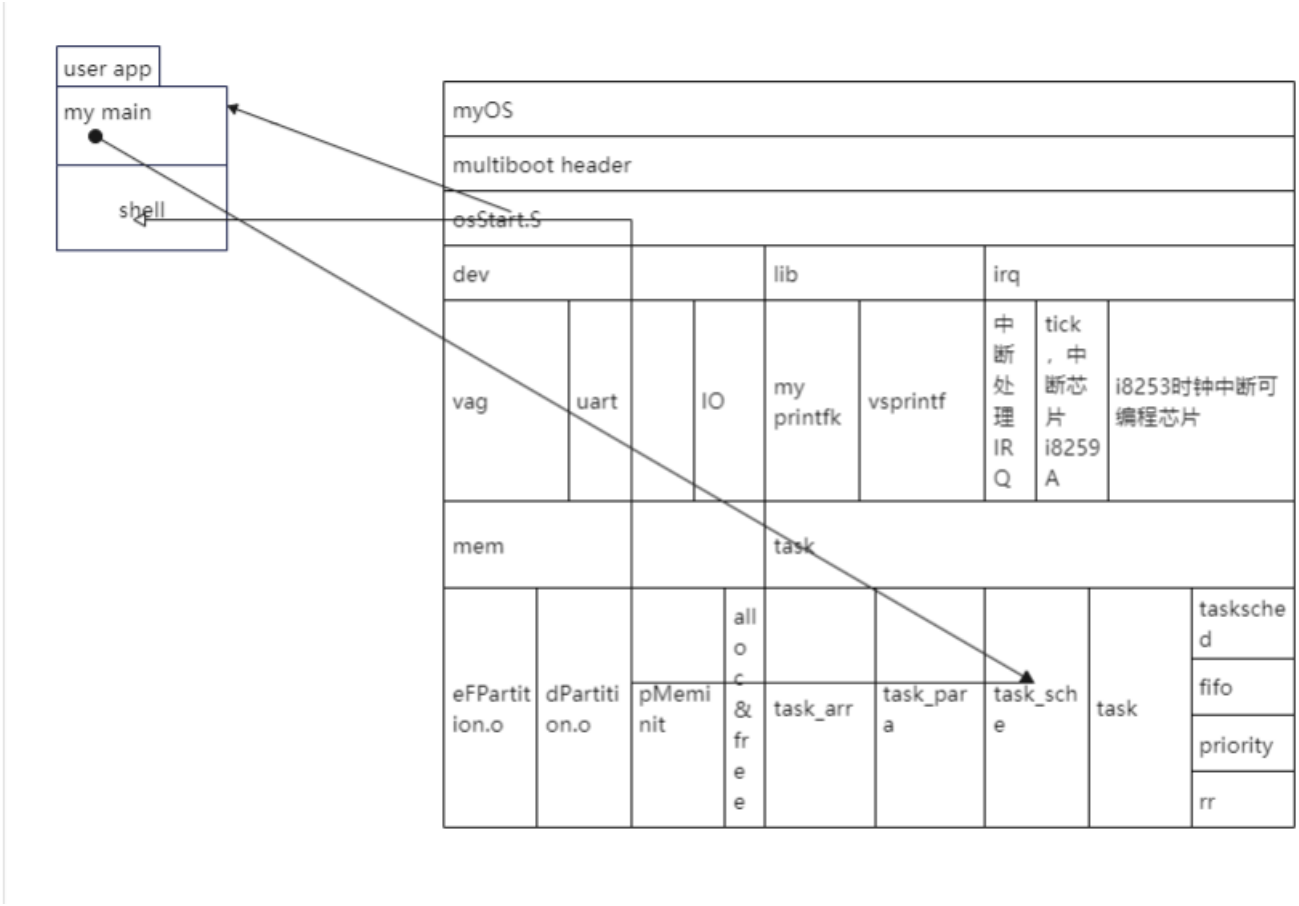
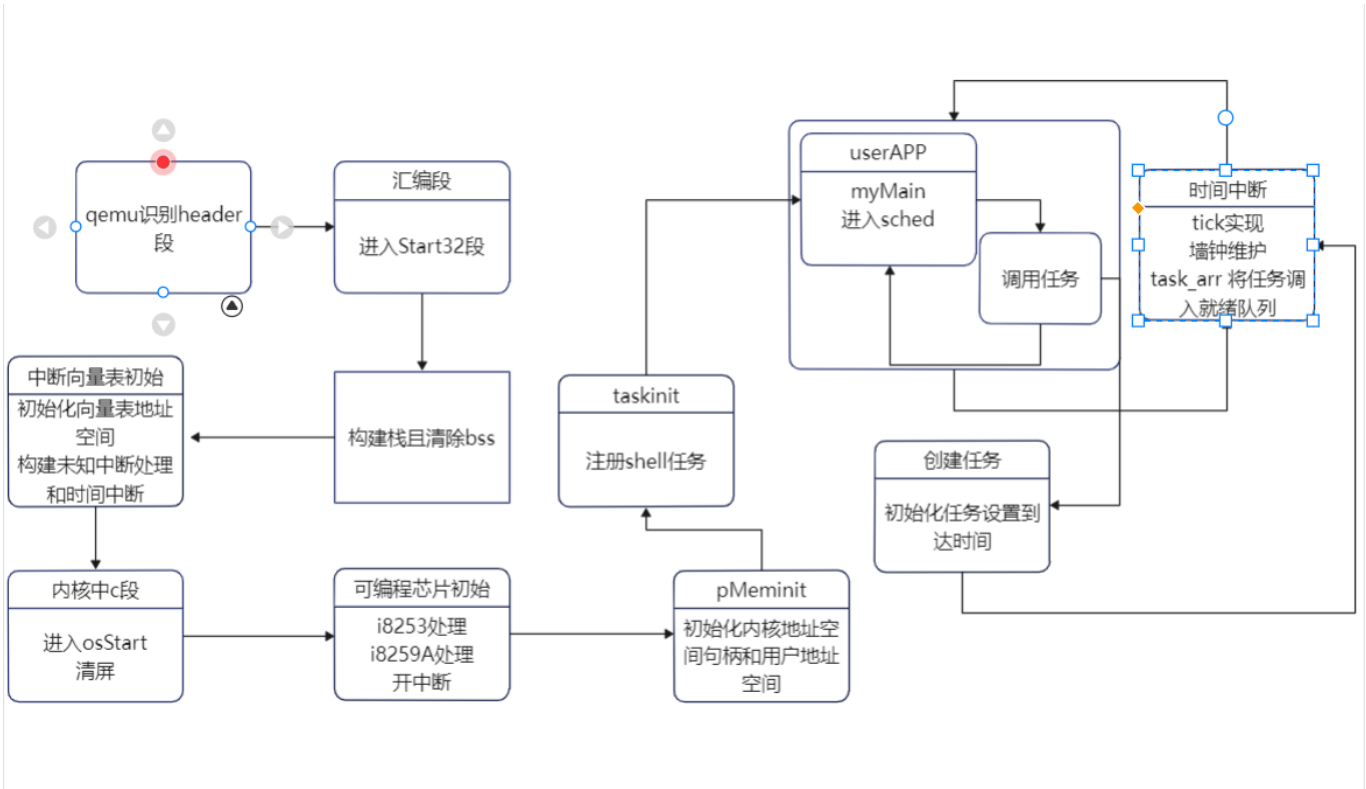
软件框图


user app
my main
shell
myOS
multiboot header
osStart.S
dev
lib
irq
vag
uart
IO
my printfk
vsprintf
中断处理IRQ
tick，中断芯片i8259A
i8253时钟中断可编程芯片
mem
task
eFPartition.o
dPartition.o
pMeminit
alloc&free
task_arr
task_para
task_sche
task
taskschedfifopriorityrr

流程实现


qemu识别header段
汇编段
进入Start32段
构建栈且清除bss
中断向量表初始
初始化向量表地址空间
构建未知中断处理和时间中断
内核中c段
进入osStart
清屏
可编程芯片初始
i8253处理
i8259A处理
开中断
pMeminit
初始化内核地址空间句柄和用户地址空间
taskinit
注册shell任务
userAPP
myMain
进入sched
调用任务
时间中断
tick实现
墙钟维护
task_arr 将任务调入就绪队列
创建任务
初始化任务设置到达时间

主要功能模块实现

注： 本代码与实例不同 本框架 更改算法与内部更改 由shell进行函数调用完成

且 shell为控制进程创建的用户接口

task.c

维护tcb （静态分配 （myTCB向量（pid即代表 myTCB向量号

栈的维护 由于栈是从高到低增长

myTCB中state 指代 （1已经就绪，0未就绪，-1处于任务池中未创建）

创建和析构仅需维护 pid值 与栈 **进程池无需初始化**；

进程池是否可用由位图维护`int ready[10]` 0为调出 1为可用

> 由findready查询

等待队列由链表waitque维护(指示等待队列， 存放到达时间)

任务随时间到达 初始state 置0；

`tick_hook_arr();` 将调用`tskStart();` 插入就绪队列

代码

```
#define NULL 0
#define IDLE 0
typedef struct myTCB{
    unsigned int pid;
    unsigned int state;
    unsigned int stack;
    tskPara para;
}myTCB;
myTCB pool[10];
int ready[10];

typedef struct waitque
{
    unsigned int pid;
    unsigned int arrtime;
    struct waitque* next;
}waitque;

waitque* head=NULL;
waitque* tail=NULL;
```

```
int findready(){
    int i=0;
    for(;ready[i]==0&i<10;i++);
    ready[i]=0;
    return i;
}// 基于位图在进程池中找到可用进程
void stack_init(unsigned int **stk, void (*task)(void)) {
*(*stk)-- = (unsigned int)0xFFFFFFFF; //栈底 kfree 基于此 free
*(*stk)-- = (unsigned int)tskEnd; // CS
*(*stk)-- = (unsigned int)task; // eip
// pushf
*(*stk)-- = (unsigned int)0x0202; // flag registers
// pusha
*(*stk)-- = (unsigned int)0xAAAAAAAA; // eax
*(*stk)-- = (unsigned int)0xCCCCCCCC; // ecx
*(*stk)-- = (unsigned int)0xDDDDDDDD; // edx
*(*stk)-- = (unsigned int)0xBBBBBBBB; // ebx
*(*stk)-- = (unsigned int)0x44444444; // esp
*(*stk)-- = (unsigned int)0x55555555; // ebp
*(*stk)-- = (unsigned int)0x66666666; // esi
**stk = (unsigned int)0x77777777; // edi
}

void createTsk(void (*tskBody)(void)){
    int cur_id=findready();//寻找可用在进程池中
    waitque *cur;
    pool[cur_id].pid=cur_id;
    pool[cur_id].state=0;
    pool[cur_id].stack=kstackalloc();// 调用专门的栈空间维护函数
    stack_init(&pool[cur_id].stack,tskBody);
    initTskPara(&pool[cur_id].para);//初始化到达时间等参数
    if(head==NULL){
        tail=head=(waitque*)kalloc(sizeof(waitque));
        head->pid=cur_id;
        head->next=NULL;
        head->arrtime=pool[cur_id].para.arrTime;
        return;
    }// 如果等待队列为空 初始化链表
    else{
        cur=(waitque*)kalloc(sizeof(waitque));
        cur->pid=cur_id;
        cur->next=NULL;
        cur->arrtime=pool[cur_id].para.arrTime;
        tail->next=cur;
        tail=tail->next;
        return;
    }//在尾部插入新任务链表
}

void destroyTsk(int tskIndex){
    ready[tskIndex]=1;//重置位图
    pool[tskIndex].state=-1;
    kstackfree (pool[tskIndex].stack);//销毁栈空间
```

```
}
void initskbody(){
    for(int i=0;i<10;i++){ready[i]=1;}//初始化进程池
    int cur_id=findready();//寻找可用在进程池中
    pool[cur_id].pid=cur_id;
    pool[cur_id].state=0;
    pool[cur_id].stack=kstackalloc();// 分配两倍的必须栈大小，函数跳转时会向栈中写入 pc
    stack_init(&pool[cur_id].stack,startShell);// 初始化shell任务
}
```

task.sched

全局变量cur_pid 指示当前进程 用于维护 调出运行时换入就绪队列 与结束时 析构进程

用于任务启动调用（sched();全部基于ctx_sw 并维护就绪队列的 换出（执行时）） 与结束 维护就绪队列 并不知道进程运行的情况，进入sched即代表一次更换进程 维护当前栈当前进程ID（栈用于进程切换，ID用于销毁`destroytask`）

换入就绪队列（被截断） 由中断程序完成

sched 对就绪队列判空 进入shell （由tsk_arr 中断跳出）

就绪队列采取存放pid和prio 的链表

tskend（进行调用 析构函数(基于cur_pid) 在轮询算法中关中断，改变watchdog 重置时间片并关闭，处理完成后，开中断，由中断进入`sched()` 或直接调用）

函数结束的处理

> 基于栈底数据返回 栈底固定为 tskEnd

```
#define FIFO 0
#define PRIO 1
#define rr 2
#define sjf 3
#define NULL 0

int sched_num=0;
rdyque *RDQtail=NULL;
rdyque *RDQhead=NULL;
int cur_pid;//记录单前pid
void sched();
//任务进入运行将其从就绪队列中调出
void queout(int pid){
    rdyque *pre;
    rdyque *cur=RDQhead;// 双指针维护链表
    if(cur->pid==pid) {
        RDQhead=RDQhead->next;
        kfree(cur);//销毁当前链表空间
        return;
```

```
    }//在头部
    for (; cur; pre=cur,cur=cur->next)
    {
        if(cur->pid==pid) break;
    }//遍历链表
    if(!cur) while (1)
    {
        myPrintk(0x4,"rdyque error");
    }//如果没有异常
    pre->next=cur->next;
    kfree(cur);//销毁当前链表空间
}
void setSysScheduler(unsigned int what)
{
    sched_num=what;
}// 设置调度算法
void tskStart(unsigned int myTCBaddr){
    myTCB *tmp=(myTCB *)(myTCBaddr);
    if(RDQhead==NULL){
        RDQtail=RDQhead=(rdyque *)(kalloc(sizeof(rdyque)));
        RDQhead->Para=tmp->para;
        RDQhead->pid=tmp->pid;
        RDQhead->next=NULL;
        tmp->state=1;
    }//若就绪队列为空，初始化
    else{
    RDQtail->next=(rdyque *)(kalloc(sizeof(rdyque)));
    RDQtail=RDQtail->next;
    RDQtail->Para=tmp->para;
    RDQtail->pid=tmp->pid;
    RDQtail->next=NULL;
    tmp->state=1;
    }//在尾部插入链表
}
void tskEnd(){
    if(sched_num==rr) {
        disable_interrupt();
        setwatchdog(0);
        enable_interrupt();
        }// 如果是轮询算法 重置时间片并关闭
    destroyTsk(cur_pid);//将进程归还进程池
    sched();//进入调度
}
unsigned int * pre_stackAddr;
unsigned int * cur_stackAddr;
void sched(){
    if(!RDQhead) {
        cur_stackAddr=pre_stackAddr;
        pre_stackAddr=&pool[0].stack;//将当前栈存入前一个栈，用作下一任务的前一个栈
        context_switch(cur_stackAddr,pool[0].stack);
    }//如果就绪队列空，进入shell
    switch (sched_num)
    {
```

```
    case FIFO :
        cur_pid=task_fifo(RDQhead);
        break;
    case PRIO :
        cur_pid=task_prio(RDQhead);
    break;
    case rr :
        cur_pid=task_rr(RDQhead);
    break;
    // case sjf :
    //     cur_pid=task_sjf(RDQhead);
    // break;
    }// 根据不同算法活得当前运行pid
    queout(cur_pid);//调出readyque
    cur_stackAddr=pre_stackAddr;
    pre_stackAddr=&pool[cur_pid].stack;//将当前栈存入前一个栈，用作下一任务的前一个栈
    setwatchdog(1);//分配时间片，其他算法不会收到影响
    context_switch(cur_stackAddr,pool[cur_pid].stack);
}
```

taskPara

```
#define PRIORITY 1
#define EXETIME 2
#define ARRTIME 3
typedef struct tskPara {
unsigned int priority;
unsigned int arrTime;
unsigned int exeTime;
} tskPara;
void initTskPara(tskPara *buffer){
    extern int hh,mm,ss;
    static int counter=0;
    buffer->priority=++counter;//将优先级与调入顺序设成相反
    buffer->arrTime=hh*3600+mm*60+ss+1;//到达时间 位当前时间加一
    buffer->exeTime=100;//100 ticks 100ms
}
void setTskPara(unsigned int option, unsigned int value, tskPara *buffer){
    switch (option)
    {
    case PRIORITY:
        buffer->priority=value;
        break;
    case EXETIME:
        buffer->exeTime=value;
    case ARRTIME:
        buffer->arrTime=value;
        break;
    }
}
void getTskPara(unsigned int option, unsigned int *para, tskPara *buffer){
```

```
    switch (option)
    {
    case PRIORITY:
        *para= buffer->priority;
        break;
    case EXETIME:
        *para= buffer->exeTime;
    case ARRTIME:
        *para= buffer->arrTime;
        break;
    }
}
```

sched

> task_sched
>
> 维护相应调度算法数据结构
>
> 各个细分算法不改动 就绪队列 只返回当前优先级最大的pid

fifo

```
void (*tick_hook)(void);
int task_fifo(unsigned int head){
tick_hook=0;//关闭hook
return *(unsigned int *) head;
}
```

prio

```
void (*tick_hook)(void);
typedef struct tskPara {
unsigned int priority;
unsigned int arrTime;
unsigned int exeTime;
} tskPara;
typedef struct rdyque{
    unsigned int pid;
    tskPara Para;
    struct rdyque *next;
}rdyque;
int task_prio(unsigned int head){
    tick_hook=0;//关闭hook

    int max,cur_id;
    rdyque *tmp=head;
    max=tmp->Para.priority;
    cur_id=tmp->pid;
```

```
    while (tmp=tmp->next)
    {
        if(tmp->Para.priority>max){
            cur_id=tmp->pid;
            max=tmp->Para.priority;
        }

    }
    return cur_id;
}
```

task_rr

> rr
>
> TSK 抢占式调度思路
>
> > 轮循算法 （时间片） 基于中断 （时间片到期）进入中断入口 在函数中 基于pid 保存至就绪队列中（tskend 不作为函数的一部分，进入即关中断，停止时间片响应，进入 sched 分配好任务参数与栈后 更新时间片开中断进入函数）
>
> 采用hook机制实现 时间片中断
>
> `*tickhook=sched_tick` 但其他机制是应把他掩蔽 `*tick_hook=0`
>
> `void tickhook(int num)` num==-1 暂停 0 关闭 1 启动 2 代表复原上次（用于暂停）将时钟暂停 以保护 printf 不被中断（确保原子性）
>
> 确保原子性的方式 暂停 时钟hook 停止计时但保留计时
>
> myPrintk 确保原子性 而 printk 保留抢占特性
>
> 只做到确保了输出的原子性，但在测试许久后会出现虽然保证原子性，但会有两次输出操作

```
int watchdog=-1;
unsigned int now=0;
int lastdog=-1;
void setwatchdog(int valid){
    lastdog=watchdog;
    if(valid==2){//暂停时间片后恢复
        watchdog=lastdog;
    }
    else{watchdog=valid;}
}//重置时间片和暂停时间片
void sched_tick(){

    static int ticks;//时间片记录
    if(watchdog==-1) return;
    if(watchdog==0) {ticks=0;}
    ticks++;
```

```
    if(!(ticks%2/*两次嘀嗒*/)){
        tskStart(&pool[cur_pid]);
        setwatchdog(0);
        sched();
    }
}
void (*tick_hook)(void);
int task_rr(unsigned int head){
    tick_hook=sched_tick;
    return *(unsigned int *) head;//返回头 ， 采用遍历的方式分配时间片
}
```

目录结构

../lab5_6/ ├── Makefile ├── multibootheader │ └── multibootHeader.S ├── myOS │ ├── dev │ │ ├── i8253.c │ │ ├── i8259A.c │ │ ├── Makefile │ │ ├── uart.c │ │ └── vga.c │ ├── i386 │ │ ├── ctx_sw.S │ │ ├── io.c │ │ ├── irq.S │ │ ├── irqs.c │ │ └── Makefile │ ├── include │ │ ├── ctx_sw.h │ │ ├── i8253.h │ │ ├── i8259.h │ │ ├── io.h │ │ ├── irq.h │ │ ├── kmalloc.h │ │ ├── malloc.h │ │ ├── mem.h │ │ ├── myPrintk.h │ │ ├── string.h │ │ ├── task_arr.h │ │ ├── task.h │ │ ├── task_rr.h │ │ ├── task_sched.h │ │ ├── uart.h │ │ ├── vga.h │ │ ├── vsprintf.h │ │ ├── wallClock.h │ │ └── watchdog.h │ ├── kernel │ │ ├── Makefile │ │ ├── mem │ │ │ ├── dPartition.c │ │ │ ├── eFPartition.c │ │ │ ├── Makefile │ │ │ ├── malloc.c │ │ │ └── pMemInit.c │ │ ├── task │ │ │ ├── Makefile │ │ │ ├── task_arr.c │ │ │ ├── task.c │ │ │ ├── taskPara.c │ │ │ ├── task_sche │ │ │ │ ├── Makefile │ │ │ │ ├── task_fifo.c │ │ │ │ ├── task_prio.c │ │ │ │ └── task_rr.c │ │ │ └── task_sched.c │ │ ├── tick.c │ │ └── wallClock.c │ ├── lib │ │ ├── Makefile │ │ └── string.c │ ├── Makefile │ ├── myOS.ld │ ├── osStart.c │ ├── printk │ │ ├── Makefile │ │ ├── myPrintk.c │ │ ├── types.h │ │ └── vsprintf.c │ ├── start32.S │ └── userInterface.h ├── source2img.sh ├── tree └── userApp ├── main.c ├── Makefile ├── memTestCase.c ├── memTestCase.h ├── shell.c ├── shell.h ├── taskTesstCase.h └── taskTestCase.c

12 directories, 69 files

makefile 组织

├── output ├── multibootheader │ └── multibootHeader.o ├── myOS │ ├── dev │ │ ├── i8253.o │ │ ├── i8259A.o │ │ ├── uart.o │ │ └── vga.o │ ├── i386 │ │ ├── ctx_sw.o │ │ ├── io.o │ │ ├── irq.o │ │ └── irqs.o │ ├── kernel │ │ ├── mem │ │ │ ├── dPartition.o │ │ │ ├── eFPartition.o │ │ │ ├── malloc.o │ │ │ └── pMemInit.o │ │ ├── task │ │ │ ├── task_arr.o │ │ │ ├── task.o │ │ │ ├── taskPara.o │ │ │ ├── task_sche │ │ │ │ ├──

task_fifo.o | | | | ├── task_prio.o | | | | └── task_rr.o | | | └── task_sched.o | | ├── tick.o | | └── wallClock.o | ├── lib | | └── string.o | ├── osStart.o | ├── printk | | └── myPrintk.o | └── start32.o ├── myOS.elf └── userApp ├── main.o ├── memTestCase.o ├── shell.o └── taskTestCase.o

地址空间

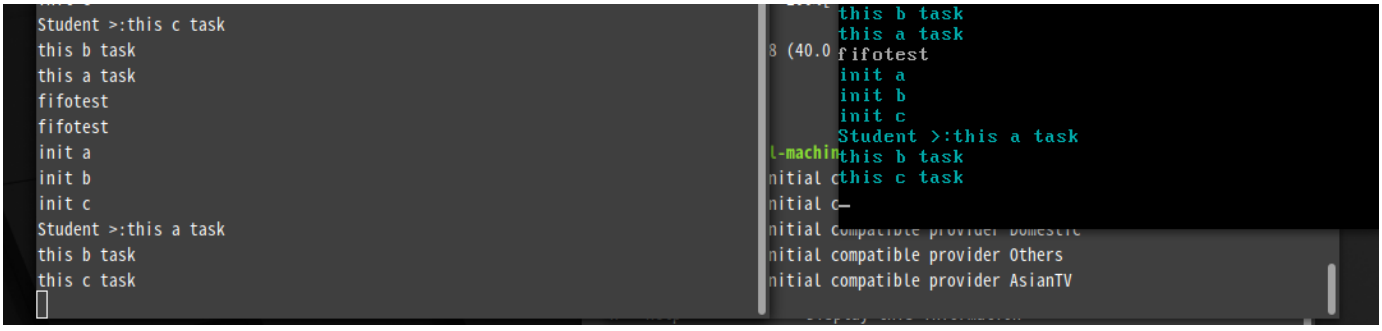
Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
.text	00100000	00105D75	R	.	X	.	L	para	0002	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	00105D75	00105D78	R	W	X	.	L	align_32	0001	public	CODE	32	0000	0000	0009	0000	0000
.rodata	00105D78	00106D41	R	.	.	.	L	dword	0003	public	CONST	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	00106D41	00106D44	R	W	X	.	L	align_32	0001	public	CODE	32	0000	0000	0009	0000	0000
.eh_frame	00106D44	00107ED0	R	.	.	.	L	dword	0004	public	CONST	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.text.__x86.get_pc_thunk....	00107ED0	00107ED4	R	.	X	.	L	byte	0005	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.text.__x86.get_pc_thunk....	00107ED4	00107ED8	R	.	X	.	L	byte	0006	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.text.__x86.get_pc_thunk....	00107ED8	00107EDC	R	.	X	.	L	byte	0007	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.text.__x86.get_pc_thunk....	00107EDC	00107EE0	R	.	X	.	L	byte	0008	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.data	00107EE0	001086F8	R	W	.	.	L	para	0009	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.got.plt	001086F8	00108704	R	W	.	.	L	dword	000A	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	00108704	00108720	R	W	X	.	L	align_32	0001	public	CODE	32	0000	0000	0009	0000	0000
.bss	00108720	00108C84	R	W	.	.	L	align_32	000B	public	BSS	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
abs	00108C84	00108C98	?	?	?	.	L	dword	000C	public		32	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...

> 后内核内存空间分配 0x1'000'000 ---0x4'000'000
>
> 用户内核空间0x4'000'000

运行结果

> 算法调度修改基于 `void setSysScheduler(unsigned int what);` 由用户（shell）修改
>
> 测试三个任务由 一个总测试任务(shell 调用)完成

fifotest

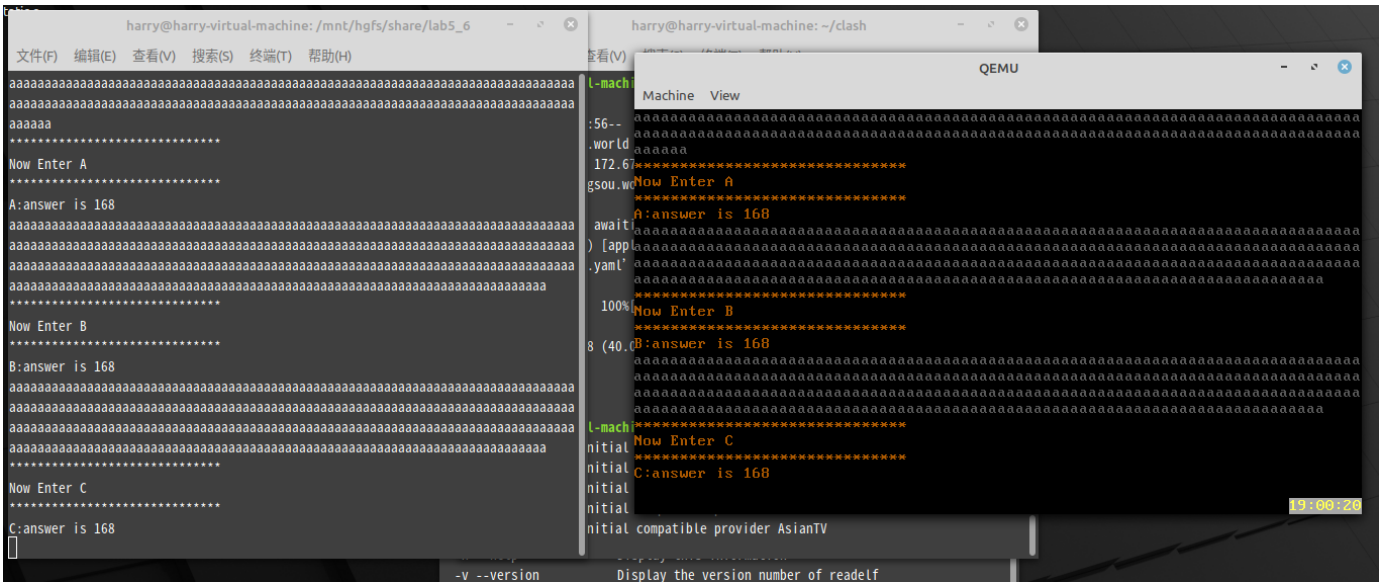


FIFO 按正确顺序输出且运行 并进入了shell 中

虽然看起来是任务在shell 出来后 运行 这是因为 任务随时间到达 由于将间隔时间设成了一秒 足以shell' 输出 用户标识， 但是 任务完成后依然 能正确识别且运行

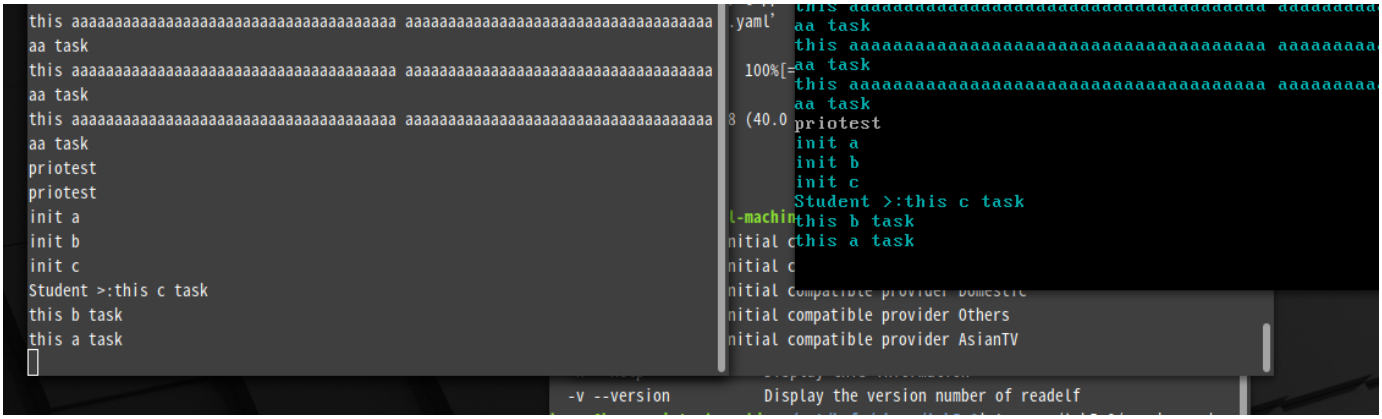
这也恰好证明了 任务随时间到达成功

test fifo



FIFO 按正确顺序输出且运行 并进入了shell 中， 计算正确

priotest

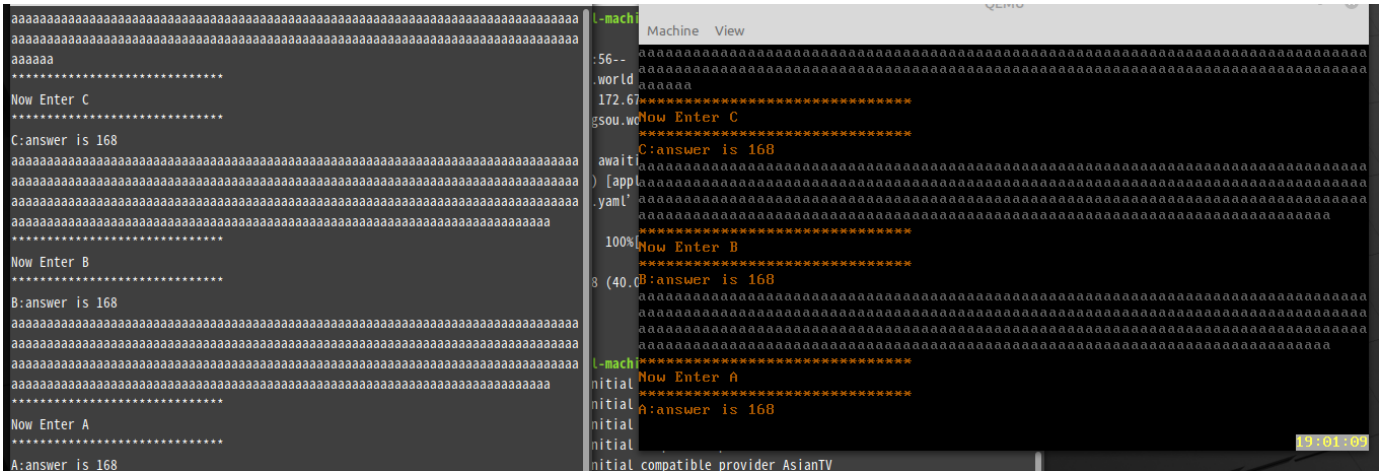


PRIO的优先级设置成了 与进入相反

init 为 abc， 调度为 cba

任务随时间到达成功

testprio



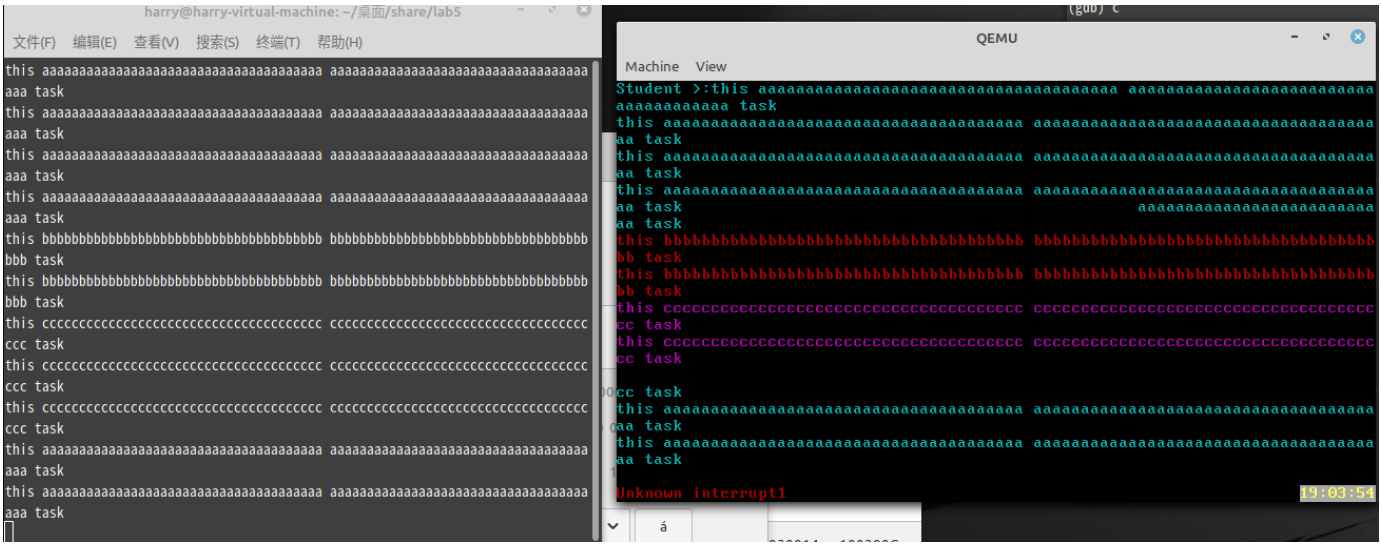
PRIO的优先级设置成了 与进入相反

init 为 abc， 调度为 cba

任务随时间到达成功

计算正确

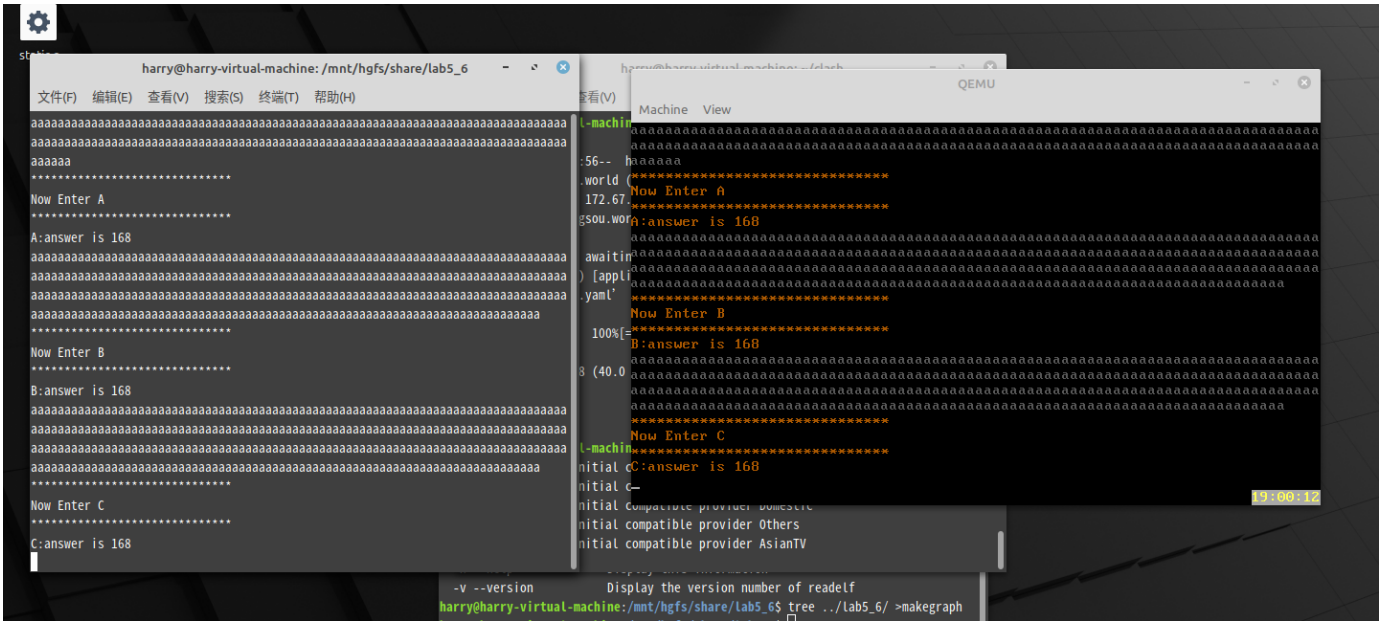
RRtest

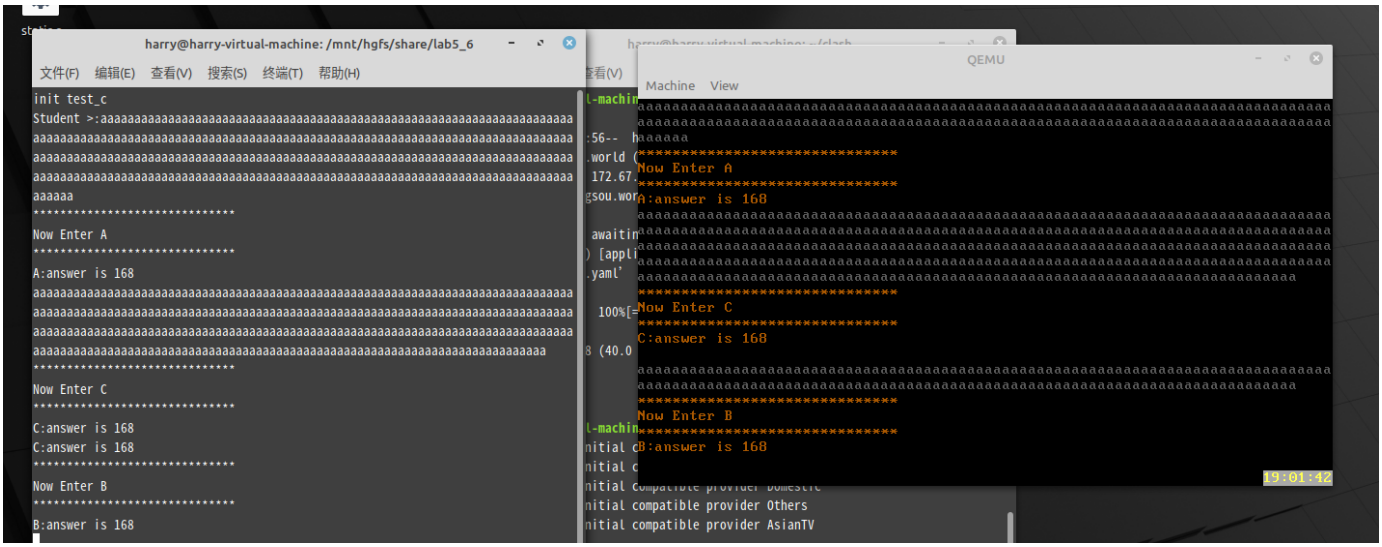


大量输出 测试中看出了 a任务由于 时间片到期被截断 实现了抢占式

任务随时间到达成功

testRR





printf的非原子性 与 myprintf占用 vga 向量 但是并不影响数值的计算 和单个myPrintf输出 计算内容为1000以内质数的数目

运算结果正确

任务随时间到达成功