lab6 实验报告

李远航

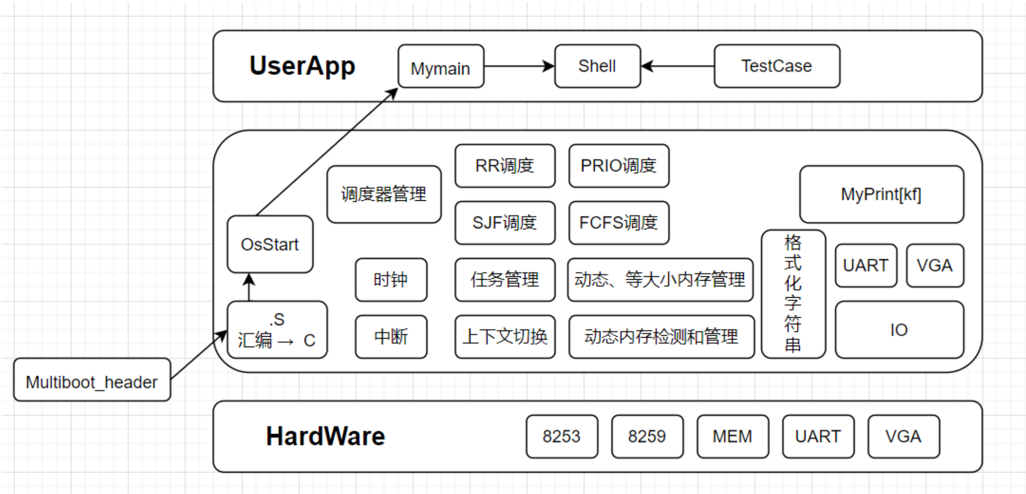
PB20000137

一、实验内容

- 【必须】调度算法，至少 2 种（不含 FCFS）
- 【根据调度算法需要修改】任务管理器
  - 【根据调度算法需要修改】任务数据结构
  - 【根据调度算法需要修改】任务创建/销毁
  - 【根据调度算法需要修改】调度器
- 【必须】自测 – 自编测试用例

二、实验原理

- 软件的架构(框图)



三、主要功能模块及其实现

- 优先队列的数据结构 以下所示为`c++`模板书写，来自数据结构大作业，具体使用时重载>``<运算符都直接换为函数，vector 替换为固定大小数组

```
//Heap.h
#pragma once
#include <bits/stdc++.h>
template <class T>
class Heap
{
private:
    std::vector<T> data;
    int length;
public:
    Heap();
    ~Heap();
```

```
    inline void swim(int k);      //上浮
    inline void sink(int k);      //下沉
    inline void push(T e);        //入堆
    inline void pop();            //出堆
    inline T top();               //返回堆顶元素
    inline bool empty();          //判断是否为空
    inline int size();            //返回大小
    inline void swap(T &a, T &b); //交换元素
};
//Heap.c
#include "Heap.h"
template <class T>
inline void Heap<T>::swap(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
template <class T>
Heap<T>::Heap()
{
    T temp;
    data.push_back(temp);
    length = 0;
}
template <class T>
Heap<T>::~Heap()
{
    data.clear();
}
template <class T>
inline bool Heap<T>::empty()
{
    return (length == 0);
}
template <class T>
inline int Heap<T>::size()
{
    return length;
}
template <class T>
inline void Heap<T>::push(T e)
{
    data.push_back(e);
    length++;
    swim(length);
}
template <class T>
inline void Heap<T>::pop()
{
    swap(data[1], data[length--]);
    data.pop_back();
    sink(1);
}
```

```cpp
template <class T>
inline T Heap<T>::top()
{
    if (!empty())
        return data[1];
    return data[0];
}
template <class T>
inline void Heap<T>::swim(int k)
{
    while (k > 1 && data[k] > data[k / 2])
    {
        swap(data[k], data[k / 2]);
        k /= 2;
    }
}
template <class T>
inline void Heap<T>::sink(int k)
{
    while (k * 2 <= length)
    {
        int j = 2 * k;
        if (j < length && (data[j] < data[j + 1]))
            j++;
        if (data[k] > data[j])
            break;
        swap(data[k], data[j]);
        k = j;
    }
}
```

- 任务随时钟动态化到达

  使用上述的数据结构，优先级比较函数如下:

```cpp
int bigger_arrv(myTCB *a, myTCB *b)
{
    return getTskPara(ARRTIME, a->para) < getTskPara(ARRTIME, b->para);
}
int smaller_arrv(myTCB *a, myTCB *b)
{
    return getTskPara(ARRTIME, a->para) > getTskPara(ARRTIME, b->para);
}
```

在时钟中断时添加 hook 函数，判读当前任务队列头元素是否可以被调度:

```cpp
void arr_hook(void)
{
    if (arrv_empty())
```

```
        return;
    myTCB *nextTask = arrv_top();
    if (get_tick_times() / 100 >= getTskPara(ARRTIME, nextTask->para))
    {
        tskStart(TCB[nextTask->tid]);
        arrv_pop();
    }
}
```

- 调度器的数据结构 建立结构体，程序初始时，根据用户选择调度算法，给不同函数指针赋值对应函数

```
typedef struct scheduler
{
    unsigned int type;
    myTCB *(*nextTsk_func)(void);
    void (*enqueueTsk_func)(myTCB *tsk);
    myTCB *(*dequeueTsk_func)(void);
    void (*schedulerInit_func)(void);
    void (*schedule)(void);
} scheduler;
void init_sch(void)
{
    switch (sch.type)
    {
    case FCFS:
        sch.schedulerInit_func = schedulerInitFCFS;
        sch.nextTsk_func = nextTskFCFS;
        sch.enqueueTsk_func = enqueueTskFCFS;
        sch.dequeueTsk_func = dequeueTskFCFS;
        sch.schedule = scheduleFCFS;
        break;
    case PRIO:
        sch.schedulerInit_func = schedulerInitPRIO;
        sch.nextTsk_func = nextTskPRIO;
        sch.enqueueTsk_func = enqueueTskPRIO;
        sch.dequeueTsk_func = dequeueTskPRIO;
        sch.schedule = schedulePRIO;
        break;
    case RR:
        sch.schedulerInit_func = schedulerInitRR;
        sch.nextTsk_func = nextTskRR;
        sch.enqueueTsk_func = enqueueTskRR;
        sch.dequeueTsk_func = dequeueTskRR;
        sch.schedule = scheduleRR;
        break;
    case SJF:
        sch.schedulerInit_func = schedulerInitSJF;
        sch.nextTsk_func = nextTskSJF;
        sch.enqueueTsk_func = enqueueTskSJF;
        sch.dequeueTsk_func = dequeueTskSJF;
        sch.schedule = scheduleSJF;
```

```
            break;
        }
    }
```

- myTCB 的数据结构

```
typedef struct tskPara
{
    unsigned int priority;
    unsigned int arrTime;
    unsigned int exeTime;
} tskPara;
typedef struct myTCB
{
    int tid;
    int status;
    unsigned long run_time;
    unsigned long this_time;
    unsigned long *stack_top;
    unsigned long *stack_max;
    tskPara *para;
    void (*function)(void);
    struct myTCB *next;
} myTCB;
```

- `task.c`内部函数具体实现
    - 上下文切换

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

    - 任务的创建和销毁

    ```
    int createTsk(void (*tskBody)(void))
    {
        if (!firstFree)
            return -1;
        myTCB *newTsk = firstFree;
        firstFree = firstFree->next;

        newTsk->function = tskBody;
    ```

```
    newTsk->stack_max = (unsigned long *)kmalloc(STACK_SIZE);
    if (!newTsk->stack_max)
        return -1;
    newTsk->stack_top = newTsk->stack_max + STACK_SIZE - 1;
    initTskPara(&newTsk->para);
    stack_init(&newTsk->stack_top, tskBody);

    return newTsk->tid;
}
void destroyTsk(int tskIndex)
{
    kfree((unsigned long)TCB[tskIndex]->stack_max);
    kfree((unsigned long)TCB[tskIndex]->para);
    TCB[tskIndex]->status = BLANK;
    TCB[tskIndex]->stack_max = 0;
    TCB[tskIndex]->stack_top = 0;
    TCB[tskIndex]->run_time = 0;
    TCB[tskIndex]->this_time = 0;
    TCB[tskIndex]->function = NULL;
    TCB[tskIndex]->next = firstFree;
    TCB[tskIndex]->para = NULL;
    firstFree = TCB[tskIndex];
}
```

- `para`相关函数

```
void initTskPara(tskPara **buffer)
{
    (*buffer) = (tskPara *)kmalloc(sizeof(tskPara));
    (*buffer)->priority = 0;
    (*buffer)->arrTime = 0;
    (*buffer)->exeTime = 0;
}
void setTskPara(unsigned int option, unsigned int value, tskPara *buffer)
{
    if (option == PRIORITY)
        buffer->priority = value;
    else if (option == ARRTIME)
        buffer->arrTime = value;
    else if (option == EXETIME)
        buffer->exeTime = value;
}
unsigned int getTskPara(unsigned int option, tskPara *para)
{
    if (option == PRIORITY)
        return para->priority;
    else if (option == ARRTIME)
        return para->arrTime;
    else if (option == EXETIME)
        return para->exeTime;
}
```

- FCFS 调度 按照时间顺序，从前往后创建任务并执行，优先级即为到来的时间，可以直接接受 arrv 队列
- SJF 调度 优先级比较函数:

```
int bigger_SJF(myTCB *a, myTCB *b)
{
    return getTskPara(EXETIME, a->para) < getTskPara(EXETIME, b->para);
}
int smaller_SJF(myTCB *a, myTCB *b)
{
    return getTskPara(EXETIME, a->para) > getTskPara(EXETIME, b->para);
}
```

- PRIO 调度 优先级比较函数

```
int bigger(myTCB *a, myTCB *b)
{
    return getTskPara(PRIORITY, a->para) > getTskPara(PRIORITY, b->para);
}
int smaller(myTCB *a, myTCB *b)
{
    return getTskPara(PRIORITY, a->para) < getTskPara(PRIORITY, b->para);
}
```

- RR 调度 添加 hook 函数，判断当前任务执行时间:

```
void RR_hook(void)
{
    if (currentTsk == idleTsk)
        return;
    if (get_tick_times() % 100 != 0)
        return;
    currentTsk->this_time++;
    if (currentTsk->this_time >= 2)
    {
        currentTsk->this_time = 0;
        if (currentTsk->run_time < getTskPara(EXETIME, currentTsk->para))
            sch.enqueueTsk_func(currentTsk);
        context_switch(&currentTsk->stack_top, BspContext); //直接调用上下文切
换返回
    }
}
```

3. 源代码组织说明

- 项目结构

```
├── 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
│   │   ├── i8253.h
│   │   ├── i8259.h
│   │   ├── io.h
│   │   ├── irq.h
│   │   ├── kmalloc.h
│   │   ├── malloc.h
│   │   ├── mem.h
│   │   ├── myPrintk.h
│   │   ├── schedulerFCFS.h
│   │   ├── scheduler.h
│   │   ├── schedulerPRIO.h
│   │   ├── schedulerRR.h
│   │   ├── schedulerSJF.h
│   │   ├── string.h
│   │   ├── taskarrv.h
│   │   ├── task.h
│   │   ├── taskPRIO.h
│   │   ├── taskQueueFIFO.h
│   │   ├── taskRR.h
│   │   ├── taskSJF.h
│   │   ├── tick.h
│   │   ├── uart.h
│   │   ├── vga.h
│   │   ├── vsprintf.h
│   │   └── wallClock.h
│   ├── kernel
│   │   ├── Makefile
│   │   ├── mem
│   │   │   ├── dPartition.c
│   │   │   ├── eFPartition.c
│   │   │   ├── kmalloc.c
│   │   │   ├── Makefile
│   │   │   ├── malloc.c
│   │   │   └── pMemInit.c
│   │   ├── scheduler
```

```
│   │   │   ├── Makefile
│   │   │   ├── scheduler.c
│   │   │   ├── schedulerFCFS.c
│   │   │   ├── schedulerPRIO.c
│   │   │   ├── schedulerRR.c
│   │   │   └── schedulerSJF.c
│   │   ├── task
│   │   │   ├── Makefile
│   │   │   ├── taskarrv.c
│   │   │   ├── task.c
│   │   │   ├── taskPRIO.c
│   │   │   ├── taskQueueFIFO.c
│   │   │   ├── taskRR.c
│   │   │   └── taskSJF.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
└── userApp
    ├── FCFSTestCase.c
    ├── FCFSTestCase.h
    ├── main.c
    ├── Makefile
    ├── memTestCase.c
    ├── memTestCase.h
    ├── PRIOTestCase.c
    ├── PRIOTestCase.h
    ├── RRTestCase.c
    ├── RRTestCase.h
    ├── shell.c
    ├── shell.h
    ├── SJFTestCase.c
    └── SJFTestCase.h
```

- Makefile 组织

```
├── MULTI_BOOT_HEADER
│   └── output/multibootheader/multibootHeader.o
└── OS_OBJS
    ├── MYOS_OBJS
```

```
│    ├── output/myOS/osStart.o
│    ├── output/myOS/start32.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/irq.o
│    │    └── output/myOS/i386/CTX_SW.o
│    ├── PRINTK_OBJS
│    │    └── output/myOS/printk/vsprintf.o
│    ├── LIB_OBJS
│    │    └── output/myOS/lib/string.o
│    └── KERNEL_OBJS
│    │    ├── output/myOS/kernel/tick.o
│    │    ├── output/myOS/kernel/wallClock.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
│    │    ├── SCHEDULER_OBJS
│    │    │    ├── output/myOS/kernel/scheduler/scheduler.o
│    │    │    ├── output/myOS/kernel/scheduler/schedulerFCFS.o
│    │    │    ├── output/myOS/kernel/scheduler/schedulerPRIO.o
│    │    │    ├── output/myOS/kernel/scheduler/schedulerSJF.o
│    │    │    └── output/myOS/kernel/scheduler/schedulerRR.o
│    │    └── TASK_OBJS
│    │         ├── output/myOS/kernel/scheduler/task.o
│    │         ├── output/myOS/kernel/scheduler/taskarrv.o
│    │         ├── output/myOS/kernel/scheduler/taskQueueFIFO.o
│    │         ├── output/myOS/kernel/scheduler/taskPRIO.o
│    │         ├── output/myOS/kernel/scheduler/taskSJF.o
│    │         └── output/myOS/kernel/scheduler/taskRR.o
│    └── USER_APP_OBJS
     ├── output/userApp/main.o
     ├── output/userApp/shell.o
     ├── output/userApp/FCFSTestCase.o
     ├── output/userApp/PRIOTestCase.o
     ├── output/userApp/SJFTestCase.o
     ├── output/userApp/RRTestCase.o
     └── output/userApp/memTestCase.o
```

4. 代码布局说明

Section	Offset (Base = 1M)
.multiboot_header	0

Section	Offset (Base = 1M)
.text	8
.data	16
.bss	16
_end	16

四、编译运行过程

直接运行脚本文件

```
./source2img.sh
```

根据提示重定向串口输入

脚本的执行:

- 编译各个文件，生成相应的 .o 目标文件
- 根据链接描述文件，将各 .o 目标文件进行链接，生成`myOS.elf`文件
- 使用 qemu，调用上一步生成的文件，进行模拟

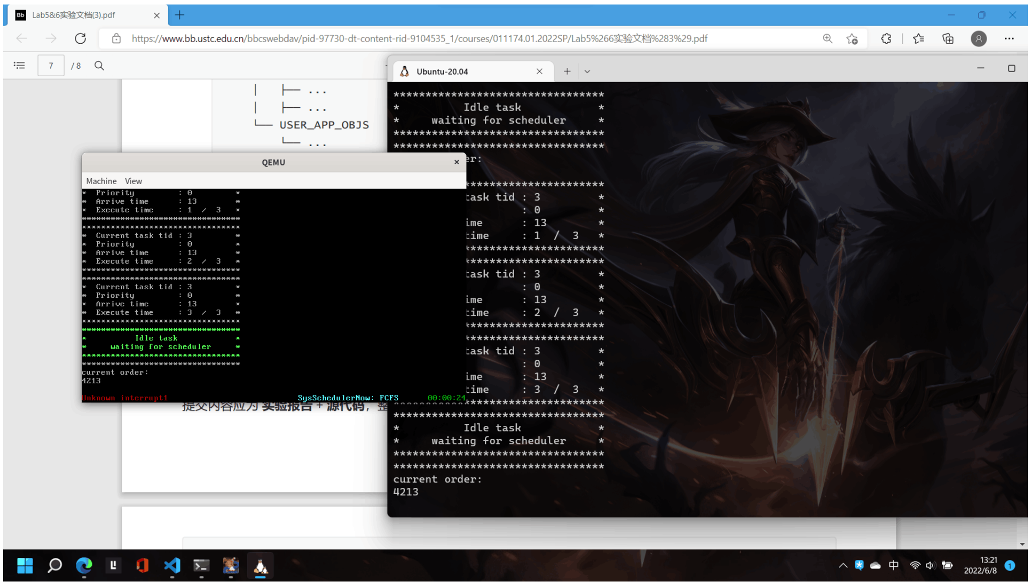
五、测试样例

> 使用 github 仓库测试样例

- FCFS

tid	prio	arrv	exe
1	0	3	4
2	0	1	5
3	0	13	3
4	0	0	2

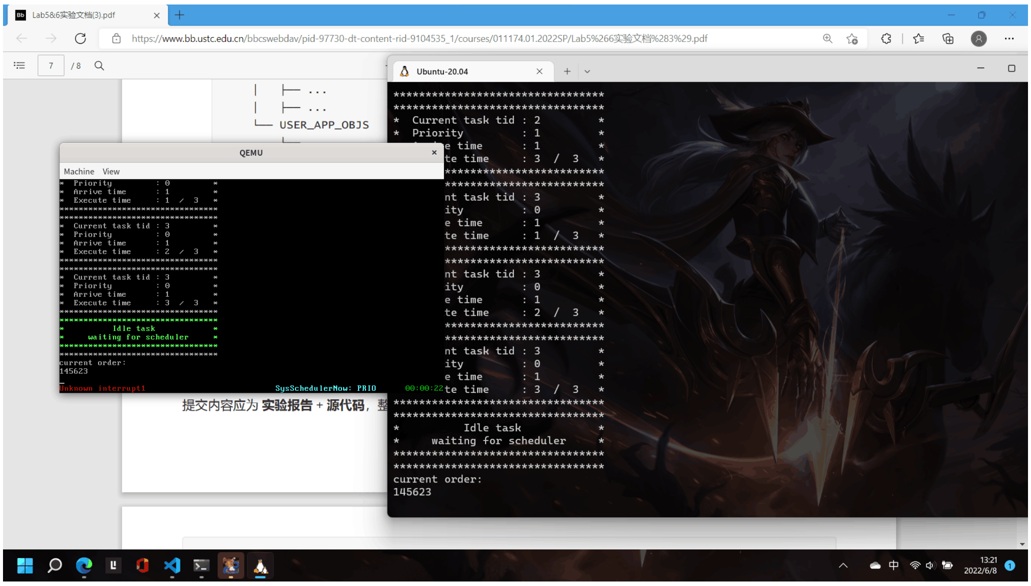
  执行的顺序应该为： `4 -> 2 -> 1 -> idle -> 3 -> idle`，与实际运行比较，符合要求



- PRIO

tid	prio	arrv	exe
1	3	0	4
2	1	1	3
3	0	1	3
4	4	1	3
5	4	4	3
6	4	6	3

执行的顺序应该为： `1 -> 4 -> 5 -> 6 -> 2 -> 3 -> idle`，与实际运行比较，符合要求

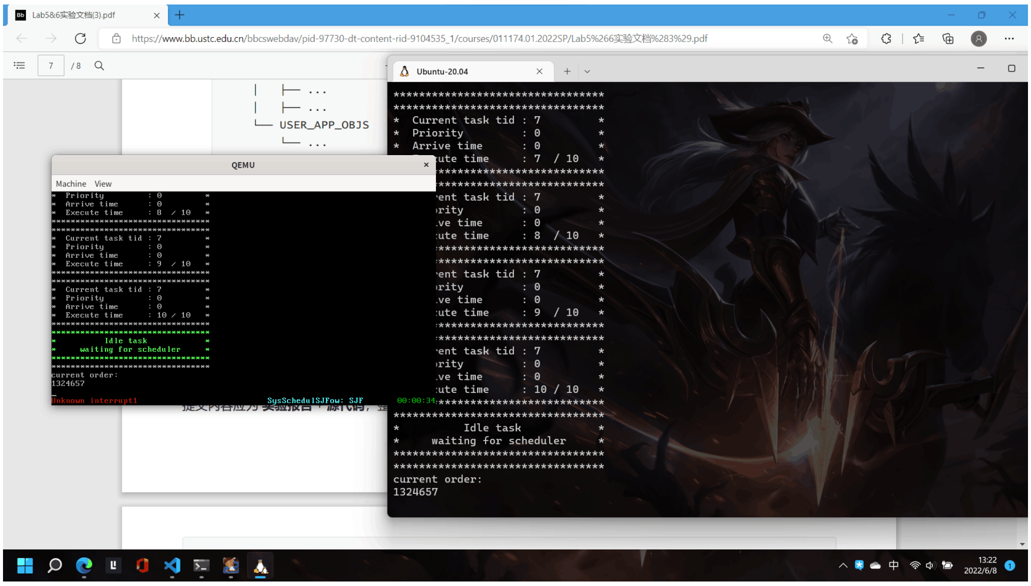


- SJF

tid	prio	arrv	exe

tid	prio	arrv	exe
1	0	0	2
2	0	0	5
3	0	0	4
4	0	10	3
5	0	12	3
6	0	11	3
7	0	0	10

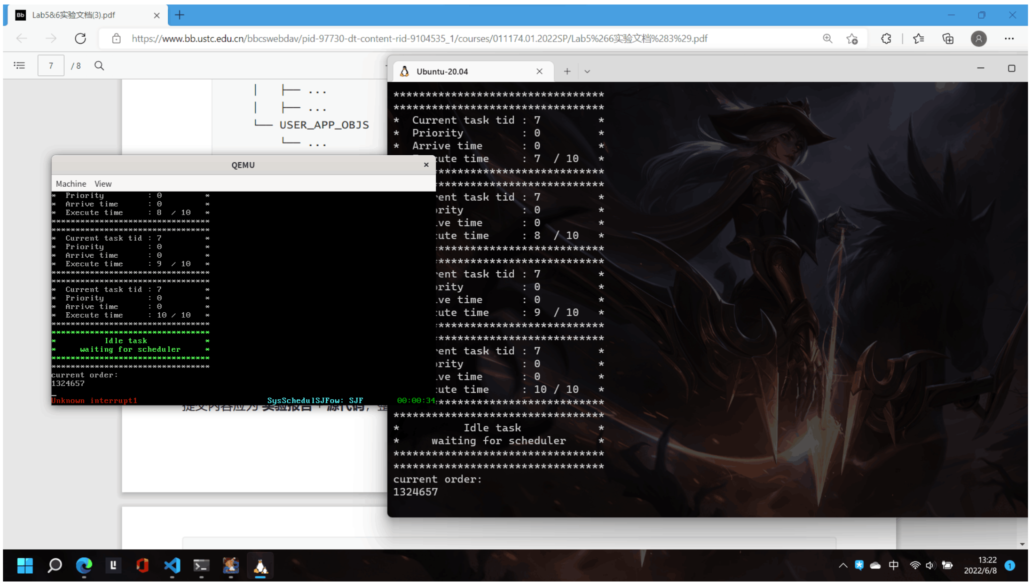
执行的顺序应该为： `1 -> 3 -> 2 -> 4 -> 6 -> 5 -> 7 -> idle`，与实际运行比较，符合要求



- RR

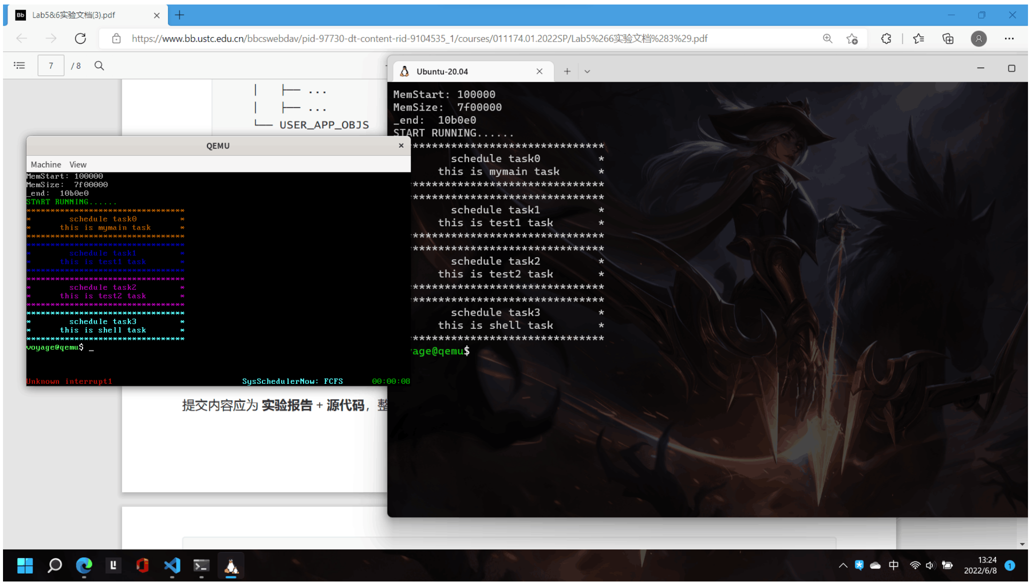
tid	prio	arrv	exe
1	0	0	14
2	0	1	4
3	0	2	4
4	0	15	3
5	0	15	4
6	0	26	4

执行的顺序应该为： `1 -> 2 -> 3 -> 1 -> 2 -> 3 -> 1 -> 1 -> 4 -> 5 -> 1 -> 4 -> 5 -> 1 -> 6 -> 1 -> 6-> idle`，与实际运行比较，符合要求



- shell

  shell 使用 FCFS 调度，测试如下：



六、实验收获

- 熟悉了操作系统不同的调度算法
- 进一步掌握了 Makefile 的组织
- 练习了 debug 的技巧
- 对 hook 函数有了更深的见解
- 遇到的问题
  - hook 函数的执行顺序可能影响最终的结果
  - 重复定义的变量需要加 extern 标识符