1. 课程设计任务、要求、目的

1.1 课程设计任务

依据操作系统课程所介绍的高级调度、中级调度和低级调度的三级调度模型，按照内核代码的实现原则，实现了高中低三级调度模型的操作系统内核模块。

1.2 课程设计目的和要求

本课程设计的目的是实现操作系统三级队列调度模型的内核模块，应包含两个部分，一个部分是按内核代码原则设计的内核模块，由一系列的函数组成；另一个部分是演示系统，调用内核模块中提供的相应函数，让其运行，同时展示该系统的运行状态，显示系统的关键数据结构的内容。

具体的要求包括：

- 建立操作系统内核中的作业调度、进程调度、中级调度的三级队列调度模型；
- 使用两种方式产生作业/进程:(a)自动产生，(b)手工输入；
- 构建作业控制块和进程控制块等核心数据结构；
- 计算并显示一批作业，从创建作业、创建进程，到作业和进程在诸队列上流转的完整过程。
- 将一批作业/进程的执行情况存入磁盘文件，以后可以读出并重放；
- 每类调度需要实现一种调度算法。

2. 开发环境

Windows 10，Visual Studio 2017

3. 相关原理及算法

3.1 调度对象

3.1.1 作业与作业步

作业是批处理系统中调入内存的基本单位，相比程序其概念更为广泛，通常还配备有相应的说明。作业步则是作业运行期间需要经历的若干个相对独立的步骤，它们之间互相依赖，顺序相关。

为了更好的管理和调度作业，多道批处理系统为为作业配备了作业控制块，其中包含了作业标识、状态、调度信息如优先级和作业已运行时间、资源需求等重要信息。

3.1.2 进程

进程实体由程序、相关数据段和进程控制块构成。创建进程的实质是创建进程控制块，相应地，撤销进程也就是撤销 PCB。

进程执行具有间断性的特点，因此也会处于多种状态之中，如就绪状态，运行状态，阻塞状态和挂起状态。其中，处于就绪状态表明进程仅分配到除 CPU 以外的所有必要资源，只要等待获得空闲处理机，便能运行。多个处于就绪状态的进程排成一个队列，即为就绪队列。挂起状态的引入是考虑到有时用户想要主动

的暂停某些正在执行或正在就绪状态的进程，可能是为了修改其中的错误，或者是调节处理机当前较为繁重的负荷。

进程控制块是系统为了描述和控制进程的运行而定义的一种数据结构，是操作系统最重要的记录型数据结构，因为其中包含了当前运行的进程及进程控制的全部信息，例如，进程标识符，处理机状态，进程状态、优先级、程序和数据段地址、同步和通信信息、资源清单等。在进程创建时要做的工作包括申请空白PCB，申请进程所需资源，初始化进程控制块，将新进程插入就绪队列，终止时则要根据标识符定位对应的PCB，从中获取进程状态、释放所占用的所有资源以及将其PCB移出队列。

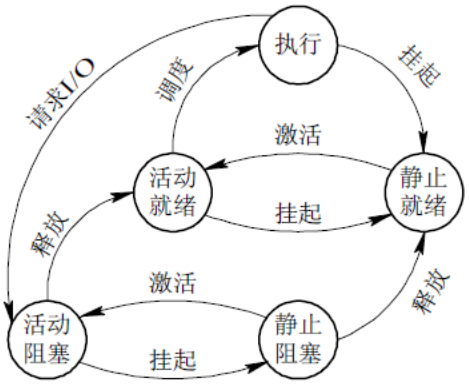


图 1 进程具有的四种状态图

3.2 操作系统处理机调度层次

3.2.1 高级调度

高级调度（High Level Scheduling），又称作业调度或长程调度，主要是根据某种作业调度算法，把处于外存上作业后备队列中的某些作业调入内存中。

3.2.2 低级调度

低级调度（Low Level Scheduling），又称短程调度或进程调度，调度的对象是进程，在这种类型的操作系统中都存在这一级别的调度。

在执行低级调度时，要完成的任务包括保存处理机现场信息，存入PCB相应单元，按某种算法选取进程，并由分派程序恢复其现场信息，为之分配处理机，从上一次运行的断点处开始继续运行。但是，当处理机切换时执行的上下文切换会占用很多处理机时间，而低级调度进行的频率较高，所以应当选择复杂度适当的进程调度算法，以及是否使用抢占方式，来尽量降低进程调度的处理机开销。

3.2.3 中级调度

中级调度（Intermediate Level Scheduling），主要作用是提高内存的利用率和系统吞吐量，为此，调度那些暂时不能运行进程到外存上去等待，以此释放宝贵的内存空资源。

3.3 操作系统三级队列调度模型

上述的三种调度中的作业队列和进程队列可组合成为调度队列模型。

进程在执行时，调度时间到达的时候可能存在任务已完成，在释放处理机后

转换为完成状态，或任务未完成，重新进入就绪队列队尾，或者是中途阻塞和被挂起，相应地分别进入阻塞队列和挂起队列。

具有高级调度的调度队列模型会先按某种算法从外存中的作业后备队列中按某种作业调度算法选择一批作业进入内存，为每一个建立进程，排入就绪队列中，接下来进行低级调度，判断哪一进程将获得处理机（针对单处理机系统）。

具有中级调度的系统会根据系统的规模设置多个阻塞队列，分别排列由于不同事件引起的阻塞进程，提高调度效率。

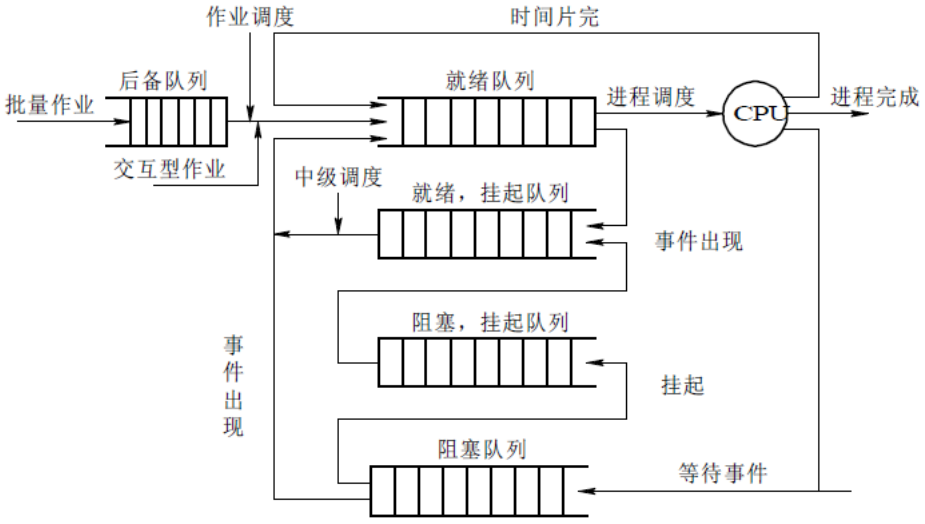


图 2 具有三级调度的调度队列模型

3.4 调度算法的选择准则

在为不同级别的调度选择相应的调度算法时，可从周转时间、响应时间、截止时间保证和优先权级准则多方面进行衡量，改善用户的使用体验，同尽可能提高系统吞吐量，处理机利用率，以及平衡各个类型的资源的利用率。

具有中级调度的系统会根据系统的规模设置多个阻塞队列，分别排列由于不同事件引起的阻塞进程，提高调度效率。

3.5 调度算法

3.5.1 高响应比优先调度算法（HRRN）

高级调度选择高响应比优先调度算法。在批处理系统中，该算法可以保证作业无论长短都能获得较为公平的调入内存执行的机会。该算法使用一种动态优先级，与作业的要求服务时间成反比，随着作业等待时间（响应时间）的增长，优先级会逐步提高。但是由于其每次调度的时候都要重新计算各个作业的相应比，会增加系统开销。其变化规律可描述如下：

$$\text{优先权} = \frac{\textit{等待时间} + \textit{要求服务时间}}{\textit{要求服务时间}} = \frac{\textit{响应时间}}{\textit{要求服务时间}}$$

3.5.2 多级反馈队列调度算法（MLFQ）

低级调度使用多级反馈队列调度算法，因为使用它可不必预先估计执行时间，可以满足各种类型进程的需要。它设置多个就绪队列，并为各个队列赋予不同的优先级，逐个降低。每一个就绪队列中按先来先服务算法调度，使用时间片轮转（RR）的方法，若在一个时间片之内没有完成任务，再次入队则加入下一级队列末尾，每一级队列的时间片长度一次增加。

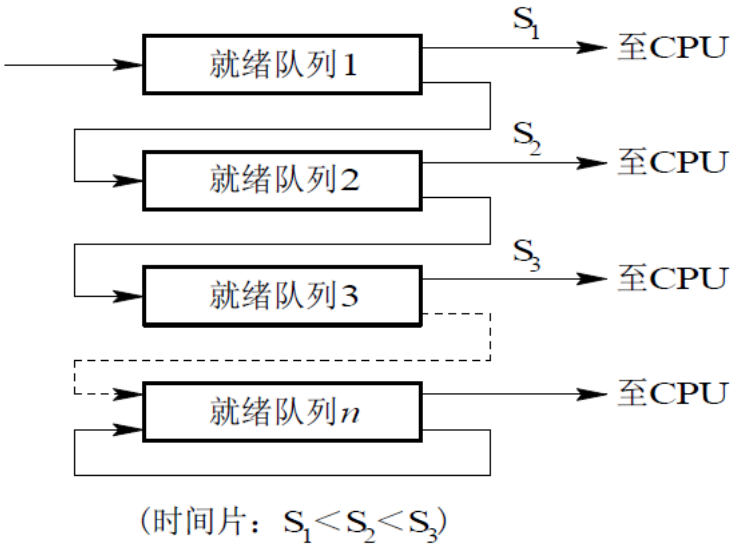


图 3 多级反馈队列算法

其中使用到的 RR 调度算法定义了一个的时间单元，称为时间片（或时间量）。一个时间片通常在 1～100 ms 之间。当正在运行的进程用完了时间片后，即使此进程还要运行，操作系统也不让它继续运行，而是从就绪队列依次选择下一个处于就绪态的进程执行，而被剥夺 CPU 使用的进程返回到就绪队列的末尾，等待再次被调度。时间片的大小可调整，如果时间片大到让一个进程足以完成其全部工作，这种算法就退化为 FCFS 调度算法；若时间片设置得很小，那么处理机在进程之间的进程上下文切换工作过于频繁，使得真正用于运行用户程序的时间减少。时间片可以静态设置好，也可根据系统当前负载状况和运行情况动态调整，时间片大小的动态调整需要考虑就绪态进程个数、进程上下文切换开销、系统吞吐量、系统响应时间等多方面因素。

3.5.3 先来先服务算法（FCFS）

处于就绪态的进程按先后顺序链入到就绪队列中，而 FCFS 调度算法按就绪进程进入就绪队列的先后次序选择当前最先进入就绪队列的进程来执行，直到此进程阻塞或结束，才进行下一次的进程选择调度。FCFS 调度算法采用的是不可抢占的调度方式，一旦一个进程占有处理机，就一直运行下去，直到该进程完成其工作，或因等待某一事件而不能继续执行时，才释放处理机。操作系统如果采用这种进程调度方式，则一个运行时间长且正在运行的进程会使很多晚到的且运行时间短的进程的等待时间过长。

4. 系统结构和主要的算法设计思路

4.1 内核模块架构

我实现的内核模块包括 kernel 和 libs 两个部分，前者包括 schedule、process 和 job 三个模块，分别实现了三级队列调度的模型，进程和作业的数据结构和调度任务，后者则定义了相关数据就结构和和一些结构体转换函数。具体构成如下

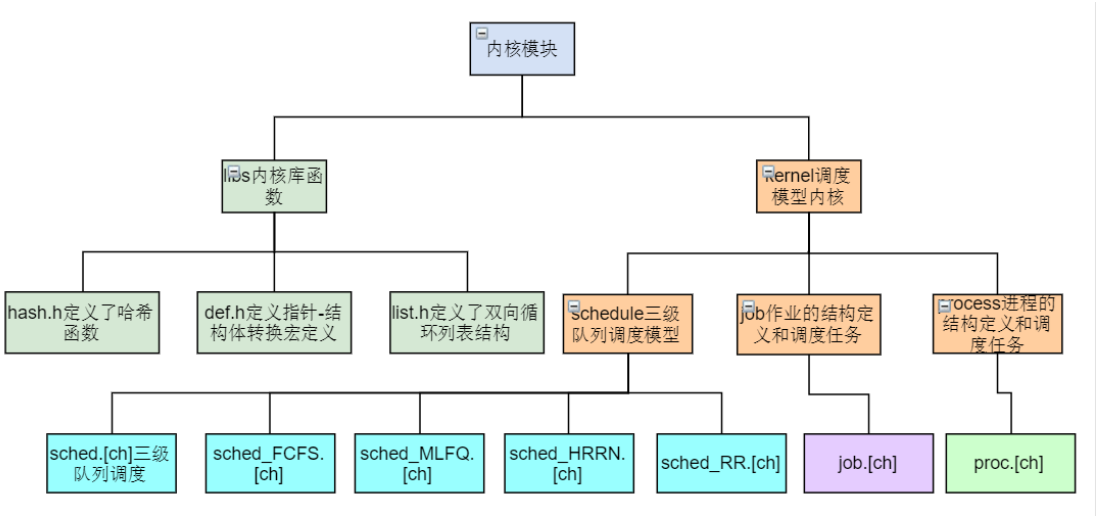


图 4 内核模块组成

由于该模型具有三级调度队列，每级调度除调度算法和对象之外基本框架一致，均有初始化、入队、出队、时间信息更新和取下一个对象的操作，所以在 kernel/schedule/sched.[ch]中实现主要的调度类框架，针对不同级别的对象另有具体的调度实例实现入口函数，分别是高级调度的 HRRN、中级调度的 FCFS 和低级调度的 MLFQ 实现，分别位于 kernel/schedule/sched_HRRN.[ch]、kernel/schedule/sched_FCFS.[ch]和 kernel/schedule/sched_MLFQ.[ch]中。其中更为具体地完成调度时的任务的函数在 kernel/job/job.[ch]和 kernel/process/proc.[ch]中定义实现，主要是针对调度对象的不同，分别进行有差异的具体操作。所有在调度模型中涉及到的队列结构，都包含了由 libs/list.h 定义的双向循环链表结构体，由它专门负责队列中前后元素的串接和查找功能。

//程序示例，调度类的指针函数定义

```
//sched.h
struct sched_class {
    const char *name;
    void(*init)(struct run_queue *rq);
    void(*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    void(*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    void(*time_tick)(struct run_queue *rq, struct proc_struct *proc);
};
//sched_HRRN.c
struct sched_class HRRN_sched_class = {
    .name = "HRRN_scheduler",
    .init = HRRN_init,
```

```
    .enqueue = HRRN_enqueue,
    .dequeue = HRRN_dequeue,
    .pick_next = HRRN_pick_next,
    .time_tick = HRRN_job_tick,
};
```

4.2 调度算法思想和实现流程

4.2.1 低级调度类设计

低级调度使用多级反馈队列调度算法，是经典的单队列的时间片轮转算法的拓展。多个 run_queue 队列确保了长时间运行的进程优先级会随着执行时间的增加而降低，而短时间运行的进程会优于长时间运行的进程被先调度执行。定义 MLFQ 总共包括 4 级队列，由 4 个双向链表组成（在 sched.c 中定义）：

```
    static struct run_queue __rq[4];
```

使用双向循环链表串接。首先初始化队列，其次，入队时判断应加入哪一个队列，由该进程是第几次入队决定，并赋予该进程相应时间片。第 i 个 rq 的最大时间片为 2 *（1<<i），其中 0<=i<4。出队时删除该进程在就绪队列中的位置。时间更新函数为当前运行进程每一时钟周期更新当前剩余时间片长度，并判断是否用完时间片而需要重新调度。具体入队流程如下所示。

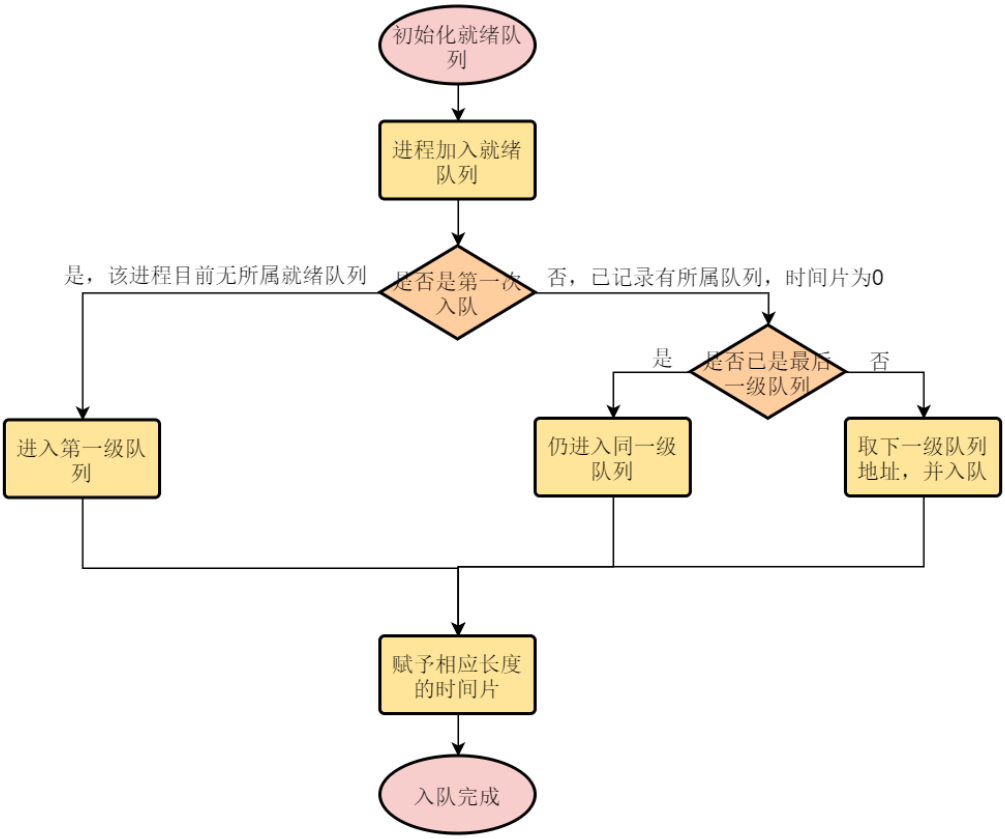


图 5 低级调度 MLFQ_enqueue 和 RR_enqueue 入队算法流程图

使用 MLFQ 算法取下一个由 MLFQ_pick_next 实现，它按顺序从 rq[0]~rq[3]

依次搜寻，直接调用 RR_pick_next 来查找某 rq 链表的头指向的就绪进程，只要找到就返回。

4.2.2 高级调度类设计

高级调度使用高响应比优先调度算法，与多级反馈队列和时间片轮转算法相比，主要不同在于每时钟周期更新作业的动态优先级，和选择下一个进行调度的方法，这时候要比较已在队列中的所有作业的优先级。具体实现如下：

```
static struct job_struct *HRRN_pick_next(struct job_queue *jq)
{
    struct job_struct *cjob, *mjob;
    struct list_ele_t *list = &(jq->job_list), *current = list_next(list), *maxi =
current;
    if (current != list) {
        while ((current = list_next(current)) != list) {
            cjob = le2job(current, job_link);
            mjob = le2job(maxi, job_link);
            if (cjob->prio > mjob->prio) {
                maxi = current;
            }
        }
        return le2job(maxi, job_link);
    }
    return NULL;
}
```

4.2.3 中级调度设计类设计

中级调度实现了 FCFS 算法的思想，本质上使用双向链表即可实现，入队 FCFS_enqueue 即为把一个就绪进程插入到就绪进程队列 rq 的队列尾，并把表示就绪进程个数的 proc_num 加 1，取下一个 FCFS_pick_next 即为选取就绪进程队列 rq 中的队头队列元素，并把队列元素转换成进程控制块指针，出队 FCFS_dequeue 即为把就绪进程队列 rq 的进程控制块指针的队列元素删除，并把表示就绪进程个数的 proc_num 减 1。以上实现完成了顺序地从就绪、阻塞或挂起队列中取出进程，将其转变为就绪挂起、阻塞挂起和就绪进程，插入目标队列的末尾候选。

4.2.4 调度类框架设计

无论对哪种调度对象，使用何种调度算法，都存在加入集合、移出集合等调度操作，可以抽象为几个基本操作，在 sched.c 分别定义为接口函数 sched_class_enqueue、sched_class_dequeue、sched_class_pick_next 和 sched_class_proc_tick，通过识别传入的调度类别标识 sched_level，采取不同动作分支，供进程、作业的调度调用。

4.2.5 建立三级调度模型

在 sched.c 中定义处理及状态更新函数 update_cpu，在系统运行期间，每

个时钟周期都会调用一次，更新正在占用处理机的进程和所有等待的作业和进程的状态，判断当前处理及是否空闲，时间片是否用完，是否需要重新调度进程或作业。调度的核心，是判断当前执行的进程 currentproc 和作业 currentjob 的 need_resched 是否为 true，若是，则应当调度。调度的时机有三种可能的产生原因：1）每周期固定更新时间片后，发现进程时间片减为 0；2）在重新调度进程时，发现就绪队列中已无等待的进程；3）某进程被挂起或某等待事件到达。

低级调度的框架由 shced.c 中的 low_level_schedule(void)实现，具体流程如下所示。

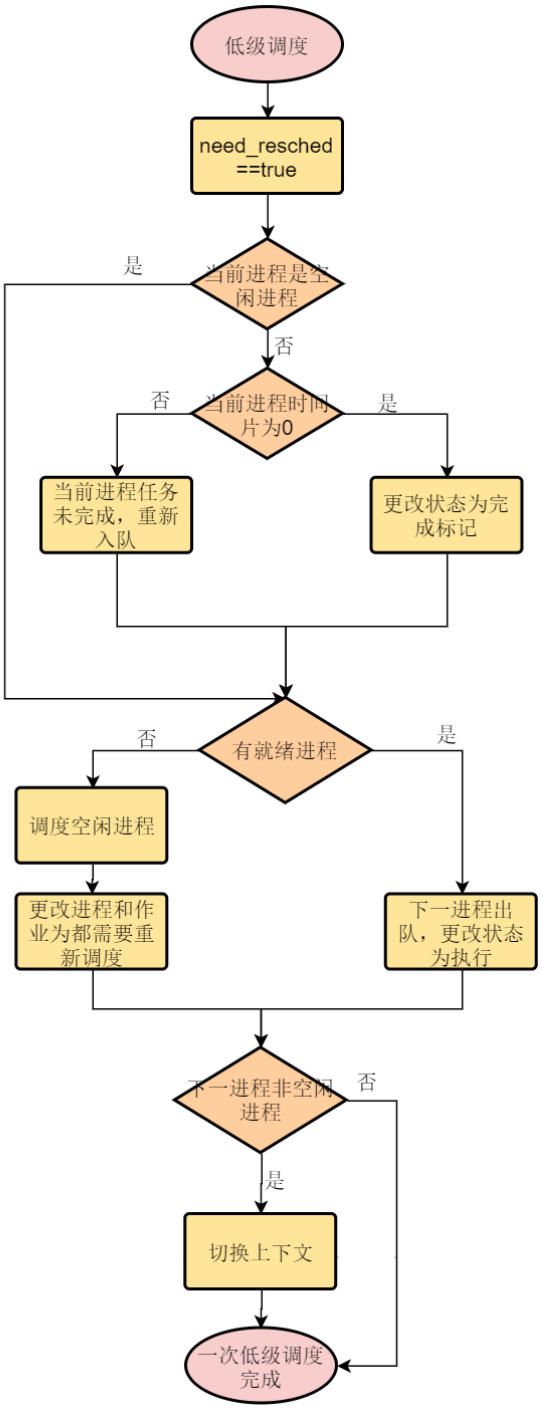


图 6 低级调度流程图

高级调度的框架由 shced.c 中的 high_level_schedule(void)实现，进行调度的时机是上一作业的所有任务完成，调入外存中响应比最高的作业：

```
    if ((next = sched_class_pick_next(LONG_TERM_SCHED)) != NULL) {
```

然后，为之创建进程，分配空白 PCB，依次进入就绪队列：

```
        currentjob->state = JOB_RUNNING;
        int i;
        for (i = 0; i < currentjob->runs; i++) {
            struct proc_struct *proc;
            if ((proc = do_fork(rq)) != NULL) {
                sched_class_enqueue(proc, NULL, LOW_LEVEL_SCHED);
            }
        }
```

具体流程图如下所示：

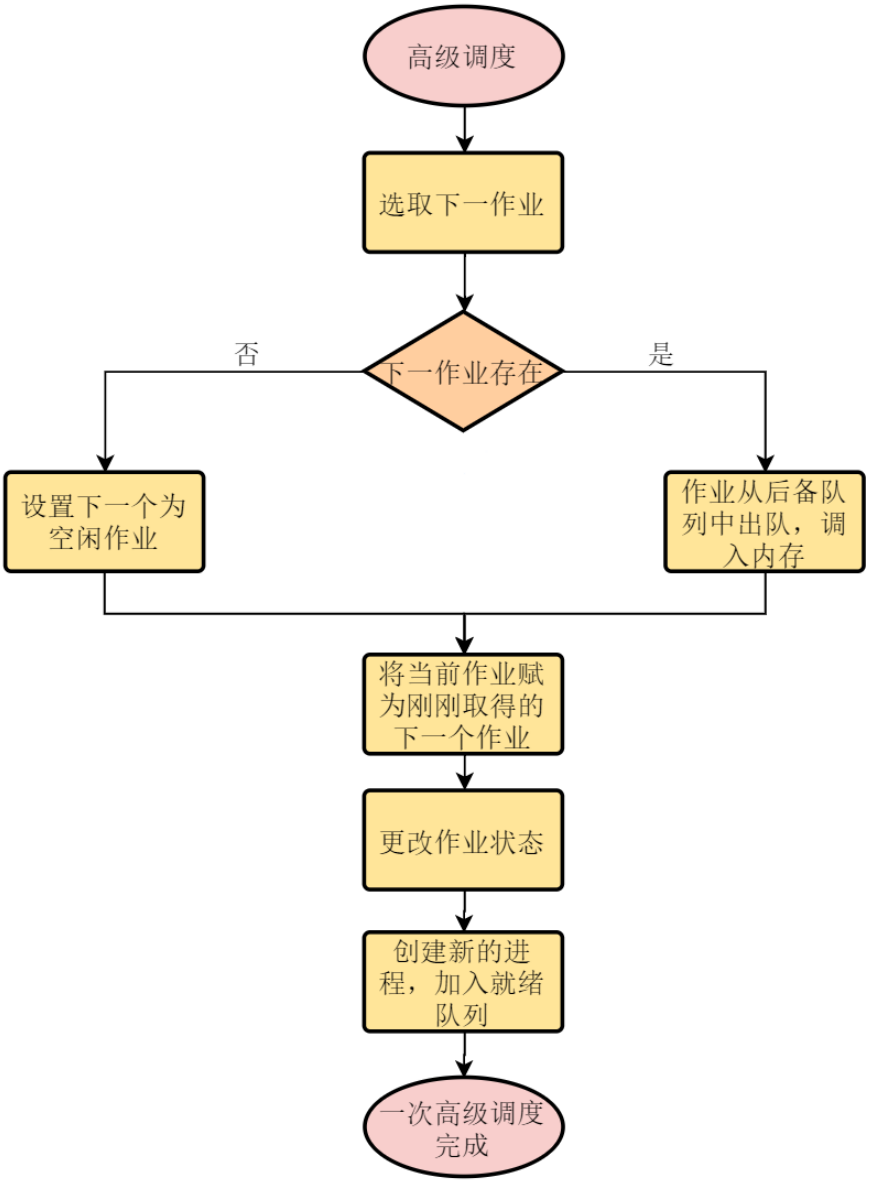


图 7 高级调度流程图

中级调度由 proc_suspend(const int pid)和 proc_active(const int pid)实现。根据进程的唯一标识符 pid 选择进程：

```
proc = find_proc(pid)) == NULL
```

分别调用入队出队的接口函数，使用 FCFS 算法放入队列，并在事件到达时恢复为就绪状态：

```
sched_class_enqueue(proc, NULL, INTERMEDIATE_LEVEL_SCHED);
sched_class_dequeue(proc, NULL, INTERMEDIATE_LEVEL_SCHED);
```

5. 程序实现---主要数据结构

本设计中设计多个组织结构，需要使用某种灵活的数据结构将相关的对量串联，因此我学习了 Linux 的实现方法，在 list.h 中定义了双向链表结构。它本身的成员只包含两个指针，没有其他个性的成员。在使用中，需要进行组织的其它结构体会将该 list_ele_t 结构体作为一个对象，用以串联前后同一类型的对象。如下图所示，展现了各种具有实际功能的结构，如 JCB、PCB 等，中间包含的链表成员是如何连接的。其中省略了纵向的其它 fields，这是每个不同结构体个性的部分。在使用中，由于我们只能获得一个结构体对象的链表指针，因此如果要根据它获得该结构体的地址，可使用 defs.h 中定义的 to_struct 和 offset 宏定义完成。

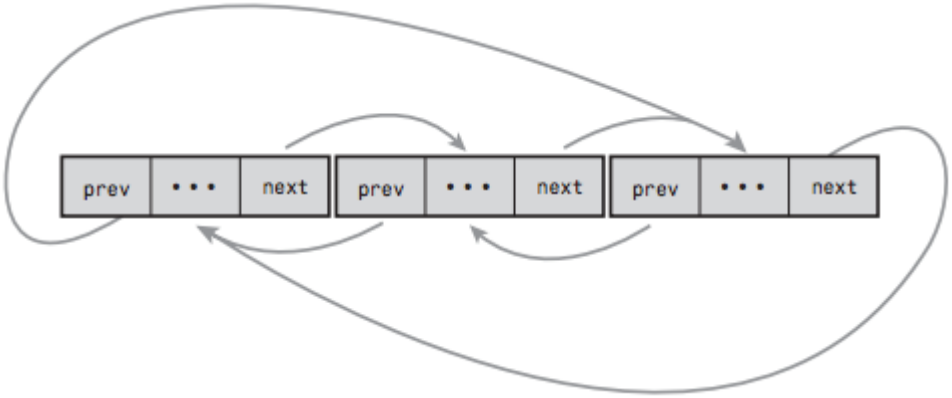


图 8 双向链表结构示意图

本设计分别为作业控制块和进程控制块设计了数据结构 proc_struct 和 job_struct。在初始化时，proc_init 和 job_init 会形成一个空闲的作业和进程。在每次产生新的作业和进程时，会通过 fork 函数进一步调用 alloc_job、alloc_proc 分配空白控制块，定义唯一标识符（get_pid），名称，资源需求和要求服务的时间信息等。

6. 程序实现---主要程序清单

list.h

list_ele_t 结构体成员	定义
struct list_ele_t *prev	向前指针

struct list_ele_t *next	向后指针

有关链表的操作

双向循环链表相关函数	定义
void list_init(struct list_ele_t *)	初始化新入口
void list_add(struct list_ele_t *, struct list_ele_t *)	在给定地址后一位添加元素
void list_add_before(struct list_ele_t *,	在给定地址前一位添加元素
struct list_ele_t *list_add_after(struct list_ele_t *, struct list_ele_t *)	在给定地址后一位添加元素
void list_del(struct list_ele_t *)	删除指定元素
void list_del_init(struct list_ele_t *)	删除指定元素并重新初始化
bool list_empty(struct list_ele_t *)	判断链表是否为空
struct list_ele_t *list_next(struct list_ele_t *)	取链表中下一个元素
struct list_ele_t *list_prev(struct list_ele_t *)	取链表中前一个元素
void __list_add(struct list_ele_t *, struct list_ele_t *, struct list_ele_t *)	在两个给定元素之间添加
void __list_del(struct list_ele_t *, struct list_ele_t *)	删除指定元素

defs.h

宏定义	定义
#define offsetof(type, member) ((size_t)(&((type *)0)->member))	求成员在结构体内部的地址偏移量
#define to_struct(ptr, type, member) ((type *)((char *)(ptr) -	由当前成员指针转换成其所在的结构体指针

offsetof(type, member)))	

sched.h

struct sched_class	定义
const char *name	调度名称
void(*enqueue)(struct run_queue *rq, struct proc_struct *proc)	入队指针函数
void(*dequeue)(struct run_queue *rq, struct proc_struct *proc)	出队指针函数
struct proc_struct *(*pick_next)(struct run_queue *rq)	选择下一个指针函数
void(*time_tick)(struct run_queue *rq, struct proc_struct *proc)	时间更新指针函数

进程就绪的组织结构

struct run_queue	定义
struct list_ele_t run_list	就绪队列
unsigned int proc_num	当前就绪进程个数
int max_time_slice	所在队列最大时间片长度
struct list_ele_t rq_link	加入某一就绪队列（共包含4个不同级别的RR队列）

后备作业的组织结构

struct job_queue	定义
struct list_ele_t job_list	后备队列
unsigned int job_num	当前在外存上的作业个数

阻塞挂起的组织结构

struct suspended_queue	定义
struct list_ele_t suspended_list	阻塞挂起队列
unsigned int suspended_num	数目

调度的级别标识

enum sched_level	定义
LONG_TERM_SCHED	高级调度
INTERMEDIATE_LEVEL_SCHED	中级调度
LOW_LEVEL_SCHED	低级调度

sched.c

调度类函数	定义
sched_init	调度类框架初始化，包括初始化各级调

	度类，后备队列，就绪队列，挂起阻塞队列
sched_class_enqueue	入队接口函数
sched_class_dequeue	出队接口函数
sched_class_pick_next	取下一个接口函数
sched_class_time_tick	时间更新接口函数

proc.h

enum proc_state 进程状态枚举	定义
PROC_UNINIT = 0	定义，但未初始化
PROC_RUNNABLE	就绪
PROC_RUNNING	执行
PROC_SUSPENDED	阻塞挂起
PROC_ZOMBIE	完成

进程控制块

struct proc_struct	定义
enum proc_state state	进程状态
int pid	作业标识符
int run_next_slice	下一次运行时间片
volatile bool need_resched	是否需要重新调度
struct mm_struct *mm	进程内存空间
char name[PROC_NAME_LEN + 1]	进程名称
struct list_ele_t list_link	进程链表
struct list_ele_t hash_link	进程哈希表
struct run_queue *rq	进程所在就绪队列
struct suspened_queue *sq	进程所在阻塞队列
struct list_ele_t run_link	进程在就绪队列中的入口地址
struct list_ele_t suspened_link	进程在阻塞队列中的入口地址
int time_slice	在当前队列中可占用处理机的时间片

进程相关函数

进程相关函数	定义
void proc_init(void)	系统初始时定义一个空闲进程，表明处理机正处于空闲状态，需要重新调度
struct proc_struct *do_fork()	创建新进程
int do_yield(void)	要求重新调度
struct proc_struct *find_proc(int pid)	通过查找存储进程标识符的哈希表进行进程搜索
void proc_run(struct proc_struct *proc)	进行进程切换上下文
static int get_pid(void)	求出一个不同的进程号作为进程的唯一标识

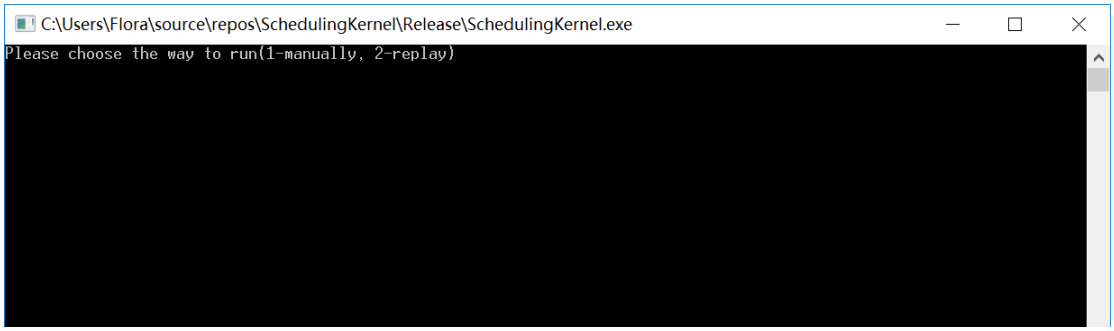
static struct proc_struct *alloc_proc(void)	空白进程控制块分配

job.c

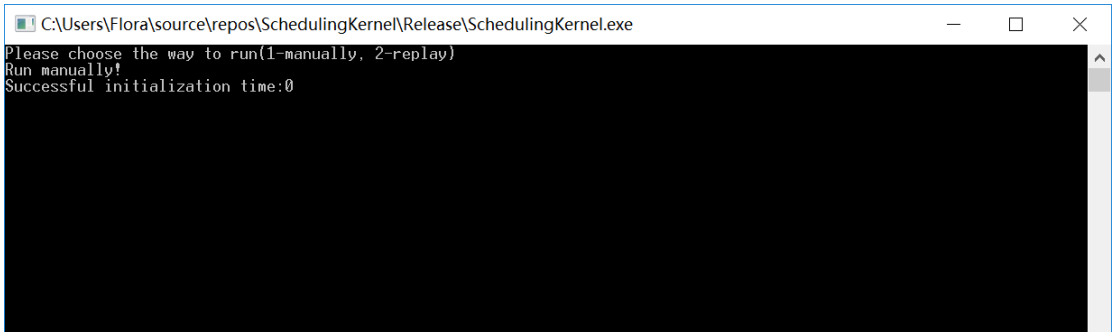
struct job_struct 作业控制块	定义
enum job_state state	作业
int jid	作业标识符
char name[JOB_NAME_LEN + 1]	作业名称
double prio	动态优先级
unsigned int estimated_time	预计要求服务时间
unsigned int waiting_time	已等待时间
volatile bool need_resched	是否需要重新调度
unsigned int runs	作业包含任务数
struct job_queue *jq	所在队列
struct list_ele_t job_link	作业在后备队列中的入口

7. 程序运行的主要界面和结果截图

初始化

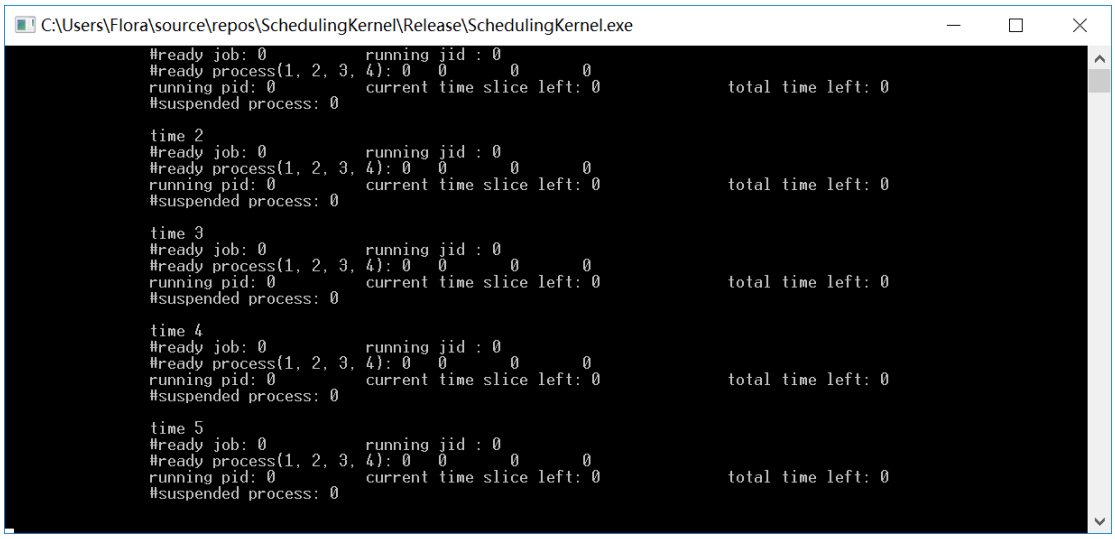


按下“1”选择功能一

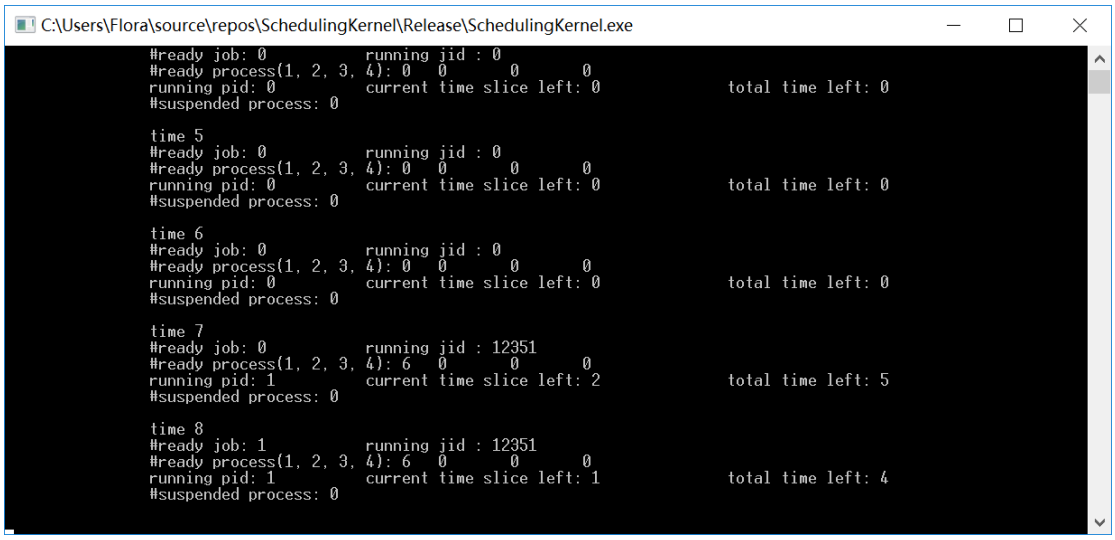


按任意键，时间流逝

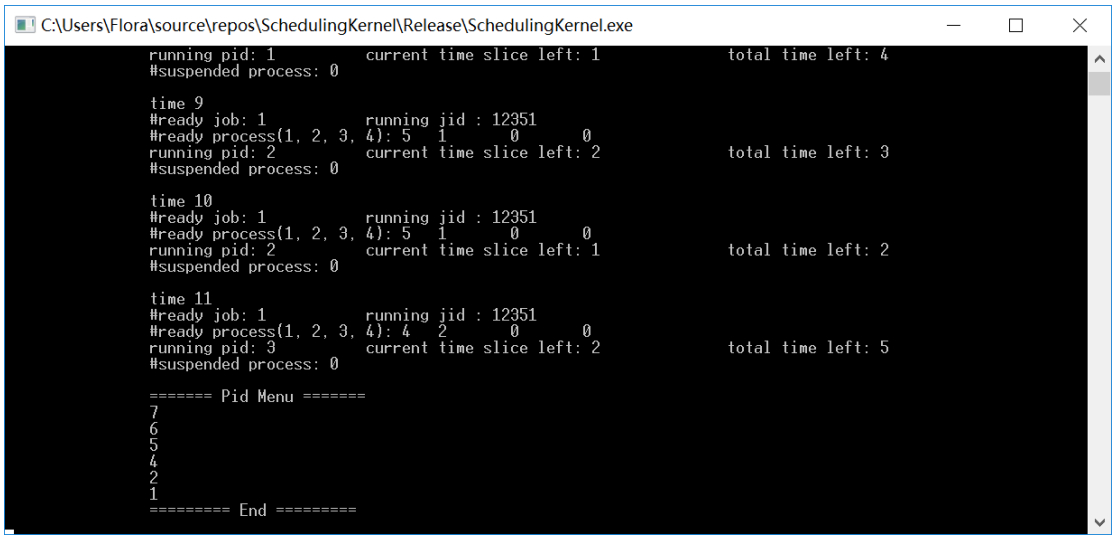
如界面所示，可以展示当前调入内存运行的作业标识符，后备队列中的作业数，多级反馈队列中各级队列上就绪的进程数，当前正在占用处理机的进程 pid，剩余时间片，挂起队列中的进程数等信息：



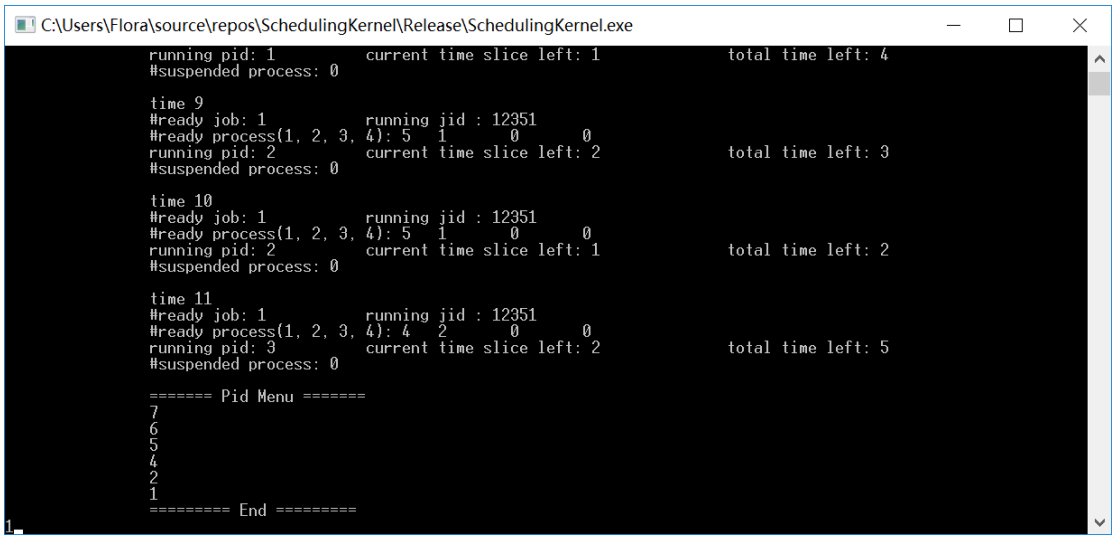
按下“1”创建作业



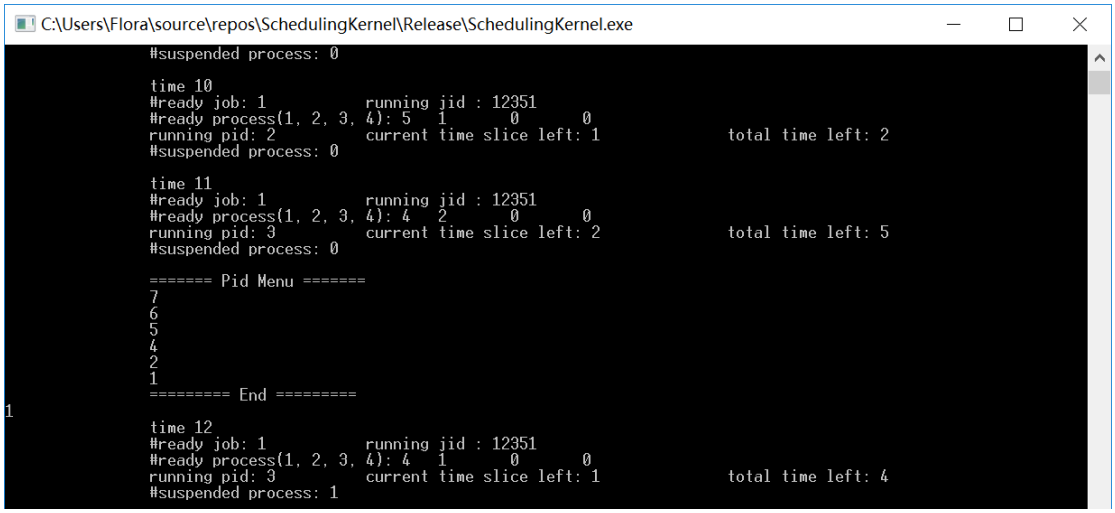
按下“-”展示所有就绪进程 pid 菜单供选择



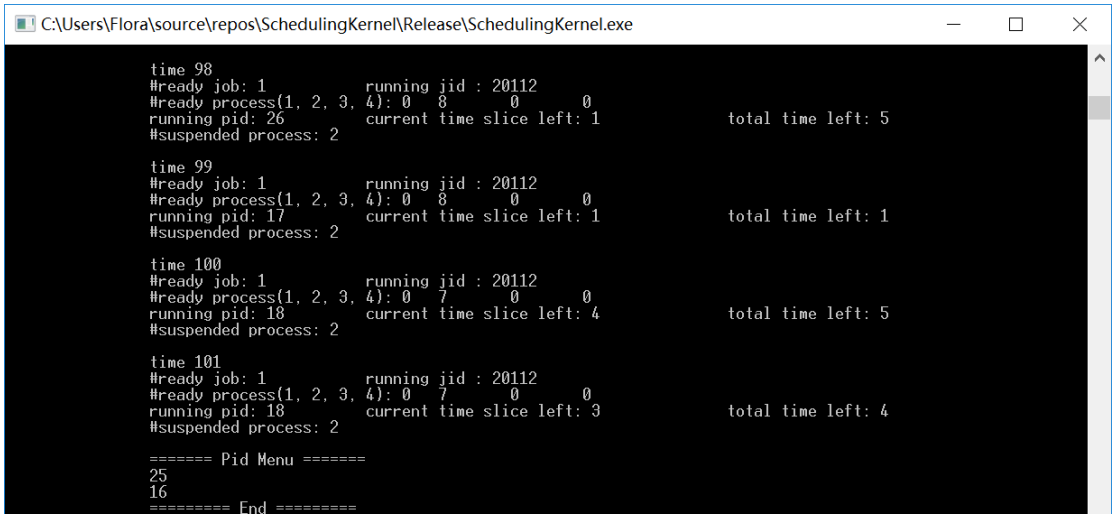
可以选中其中的一个数字按下，就可以挂起相应进程，也可以选择不包含在内的序号，不会做任何动作。这里选择挂起 1 号进程



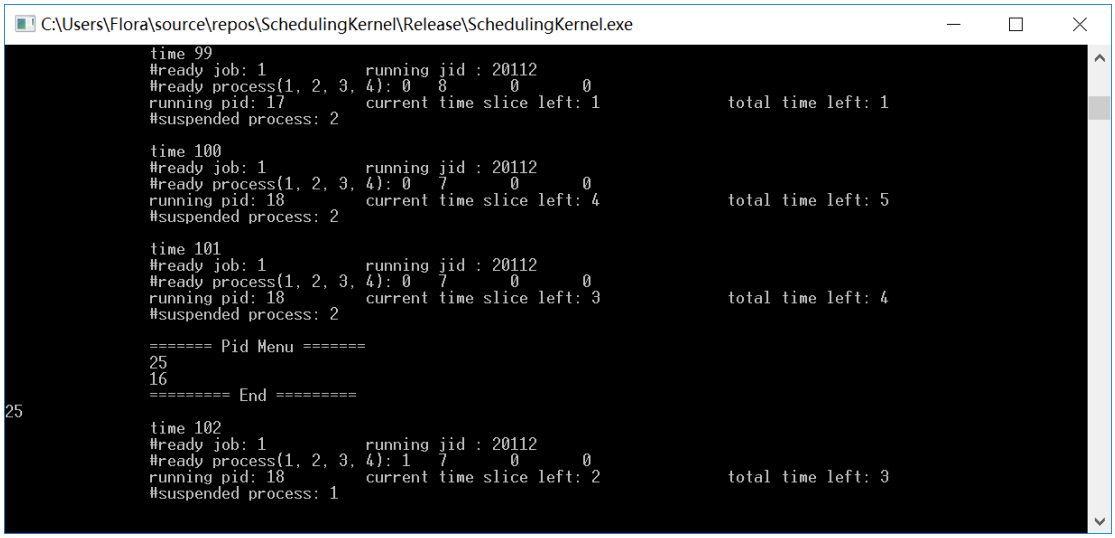
可以看到挂起后就绪和挂起队列的变化，多级队列中进程在各队列上的流转



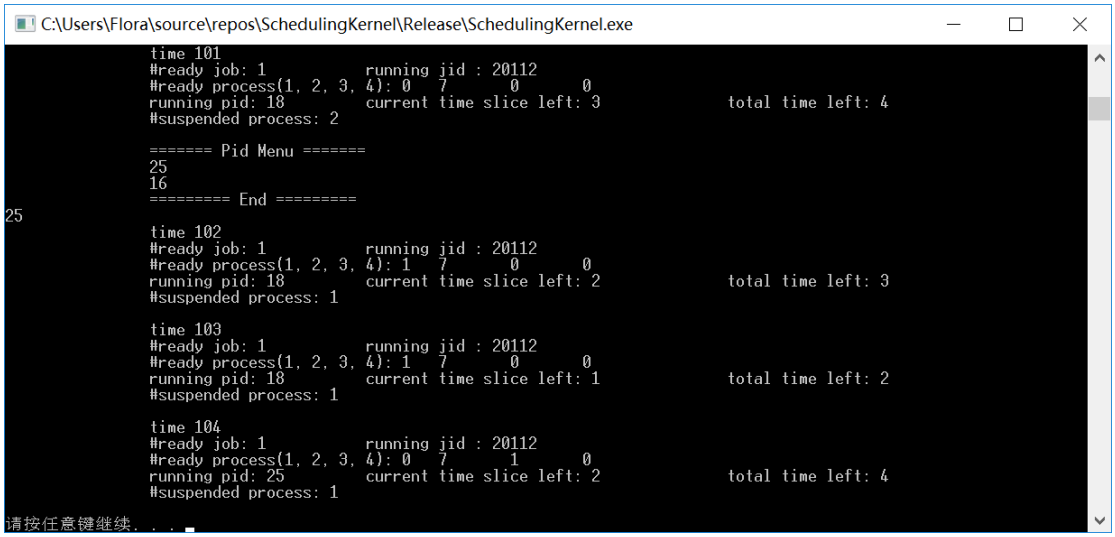
按下“+”可以也可以弹出 pid 菜单界面



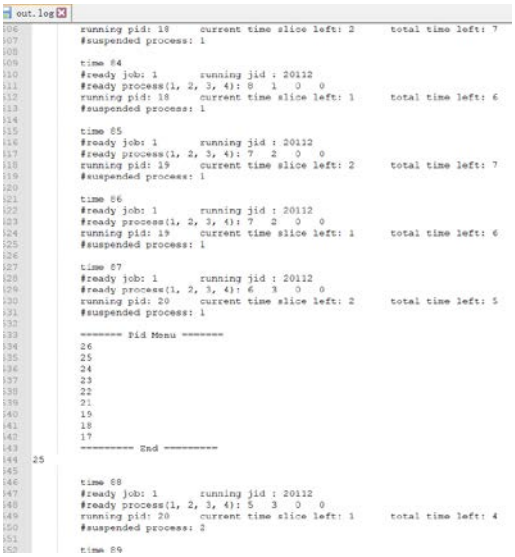
从中选择一个重新调入就绪队列



若需要退出系统，可以按下“Esc”



结束运行后，可在程序所在路径 **out/out.log** 获得输出的 **log** 文件



若开始选择功能“2”，重放，便可以直接在屏幕上打印出之前的 log 文件中保存的调度历史信息。

8. 总结和感想体会

通过完成本次课设，复习巩固了操作系统给中有关进程、作业、处理机调度、调度算法等一系列理论知识。同时，使用 C 语言编写内核模块，也帮我很好地复习了 C 语言的编写，尤其是结构体和指针的编写技巧，有点像面向对象编程，用 C 语言完成格外有趣，确实十分强大。并且，为了更好地完成本次课设，之前花了大量时间研究了 Linux Archive，尤其是 kernel 部分的代码，和核心数据结构 list 的代码，这都与本次实验密切相关。在学习其中有关 scheduling 的实现时，目前的 Linux 使用的是完全公平 CFS 调度类，基本数据结构是红黑树，可以通过权重分配的更好地平衡各个进程占用处理机的机会。同时，我还学习了其它一些小型类 Unix 系统的实现，发现，当前“作业”的概念提的比较少了，似乎都用“进程”来替代了。还有一点比较有趣的发现是，Linux 中为了同时实现多种调度算法，而又不需要每次重写相同的基本动作，它通过定义接口，使用指针函数的方式实现了一个调度框架类，类似 C++中断抽象类，纯虚函数。

参考文献

[1]汤晓丹，汤子瀛等.计算机操作系统[M]第三版.西安电子科技大学出版，2007
[2]陈渝.操作系统简单实现与基本原理.https://legacy.gitbook.com/@chyyuu
[3] Linux 内核源码.https://www.kernel.org/