目录

基于 Android/Java 处理机调度算法模拟实现

软件工程 2003 杜睿

1 设计目的

CPU 是计算机系统的重要资源，处理机调度是操作系统的核心内容。进程调度和作业调度作为处理机调度的重要组成部分，其调度算法优劣直接影响操作系统的性能好坏。模拟设计进程调度和作业调度算法，在掌握各种调度算法的同时，可以进一步加深理解处理机管理技术的原理及特点。

2 系统设计

2.1 设计任务[1]

1. 可任意选择作业（进程）数量、作业（进程）优先级、作业提交时间（进程到达时间）和预计运行时间。

2. 模拟实现作业调度中先来先服务（FCFS）、短作业优先（SJF）和最高响应比调度（HRRN）算法。每次作业调度时，显示各作业的执行情况（开始执行时间，结束时间，周转时间）；最后，计算并列出平均周转时间并对相同情况下不同调度算法进行性能分析比较。

3. 模拟实现进程调度中基于优先级的时间片轮转调度算法（PRR）和多级反馈队列轮转调度算法（MFQ）。每次进行进程切换时，显示各进程执行情况（剩余时间，进程状态，进程排队情况），最后列出各进程的开始执行时间和结束时间。

[1]基于《2020 级操作系统课程设计任务书》，2023 年 1 月，有删节

2.2 任务分析

一个作业从用户提交开始到占有处理机被执行，一般来要由系统三级调度才能实现，即作业调度、内存调度、进程调度。其中最为重要的：

作业调度 主要是完成作业[2]从后备状态到可执行状态的转变，即按照某原则，从外存后备队列中挑选作业，将其装入内存并为其创建进程；

进程调度 主要是完成进程从就绪状态到执行（完成）状态的转变，即按照某策略，从内存就绪队列中选取进程，为其分配处理机使其运行。

表 1: 常用的处理机调度策略

算法名称	主要适用范围	默认调度方式
先来先服务	作业调度 & 进程调度	非抢占式
短作业（进程）优先	作业调度 & 进程调度	非抢占式
高响应比优先	作业调度	非抢占式
时间片轮转	进程调度	抢占式（不抢时间片）
多级反馈队列	进程调度	抢占式（抢占时间片）

* 调度策略也就是调度算法

基于设计内容要求及上述理论常识，可以做出如下基本假设：

1. 作业调度和进程调度是互补共存的层级关系。不考虑内存调度的前提下，实现任何算法，至少要实现两级调度：作业调度及进程调度。
2. 作业和进程可以合二为一，统一抽象为任务[3]。不考虑阻塞的前提下，一个任务大致可归为未提交态、收容态、就绪态、运行态及销毁态。
3. 调度是种将何资源分配给何任务的决策行为。从各级调度的交互上说，调度可以分为：本级进行调度、移交上层调度及委托下层调度。

综上，我对《任务书》要求的系统设计内容进行重述：通过编程，模拟实现一个单（多）道批处理系统及其多级调度策略，输入作业序列，输出作业进入内存的时间、结束时间、平均周转时间，展示各作业的执行时间线。
任务书中要求的原任务与多级调度实现的对应关系如下：

[2]用户向计算机提交任务的任务实体，《计算机操作系统教程（第四版）》，清华大学出版社
[3]作业是提交层面的任务，进程是执行层面的任务，作者注

表 2: 批处理系统的多级调度明细

原任务 *	作业调度	进程调度	内存限制
先来先服务	*	先来先服务	1
短作业优先	*	先来先服务	1
高响应比优先	*	先来先服务	1
基于动态优先级的时间片轮转	先来先服务	*	INF
多级反馈队列	先来先服务	*	INF

2.3 概要设计

要模拟批处理系统，就须要模拟时钟、任务及其状态、分级调度。

时钟 标定时间流逝的基准，以便执行作业调度、进程调度、性能分析等。每流逝一个单位时间，系统至少要进行一次调度。所以，可以建立“时钟”与“调度器”之间的依赖关系，当时钟发生改变时应该自动通知调度器，调度器接受信息并做出响应。时钟，被称为观察目标；调度器，被称为观察者。一个观察目标可以对应对个观察者，可以根据需要增加和删除观察者。这种设计模式被称为*观察者模式*或*发布-订阅模式*。

任务 每个任务都有提交时间、到达时间等属性；运行中任务的剩余时间随时间流逝而减少（从这种意义说，运行中的进程应当观察时间流逝，并自主改变剩余时间字段）。

任务揉杂作业和进程，集成两者的属性，即 id、优先级、状态、时钟，提交时间、预计需要时间，到达时间、开始时间、剩余时间、完成时间。

在模拟逻辑上，状态包含五种类型：未提交、收容、就绪、运行、结束。未提交，代表用户输入作业序列中的提交时间晚于当前时钟时刻；收容，代表作业已提交，进入外存收容队列（外存一般没有容量限制）；就绪，代表进程已创建，进入内存就绪队列（内存一般有容量限制）；运行，代表进程获得处理机资源，随时间流逝剩余时间递减。

调度 任务状态由调度器进行切换；调度是分层级的。

作业提交与调度细节应当是解耦的，同时为较好地实现“分级”调度，可以将调度器链式组织，各司其职，又保证了作业在各级调度中的传递，直到进程销毁并记录日志。这种思想来源于*职责链模式*。

机器 整合时钟、任务、调度，并接受测试样例或用户输入，输出日志。不同的调度策略在各层级上排列组合，可以衍生出多种具体“机器”。为将对象创建和对象使用解耦，可以使用*工厂方法模式*。

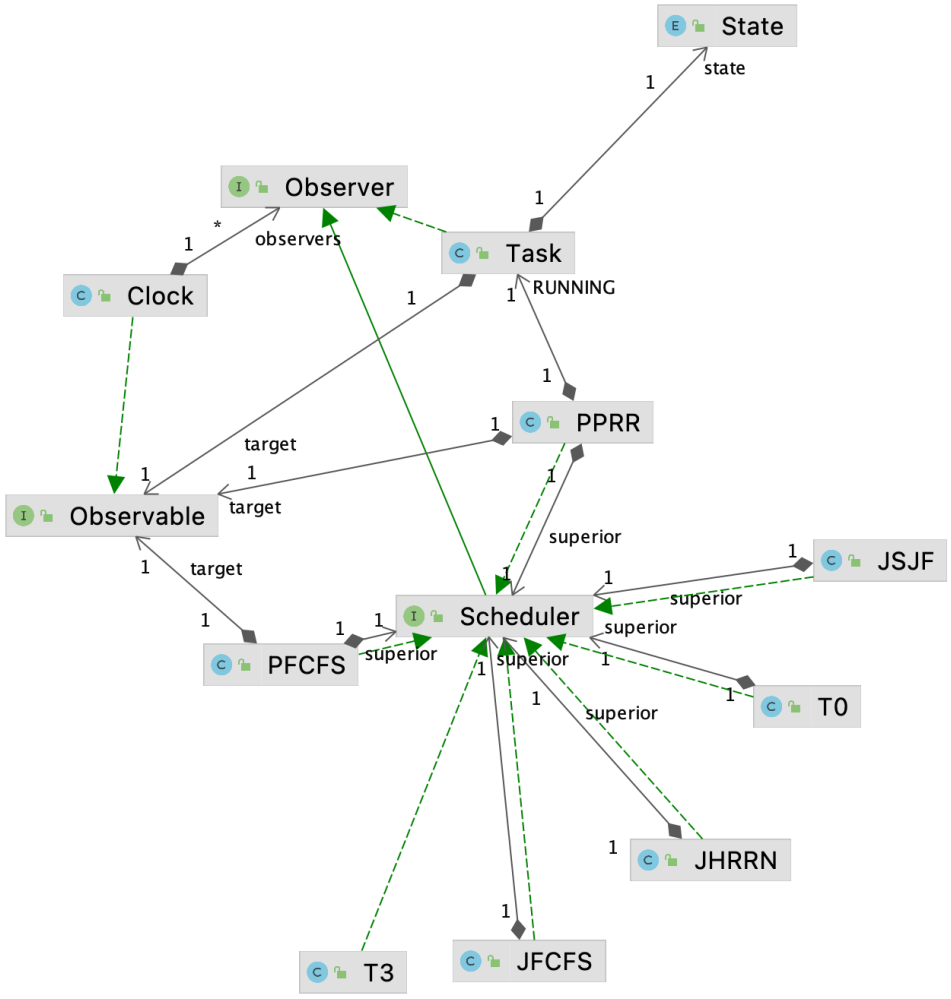
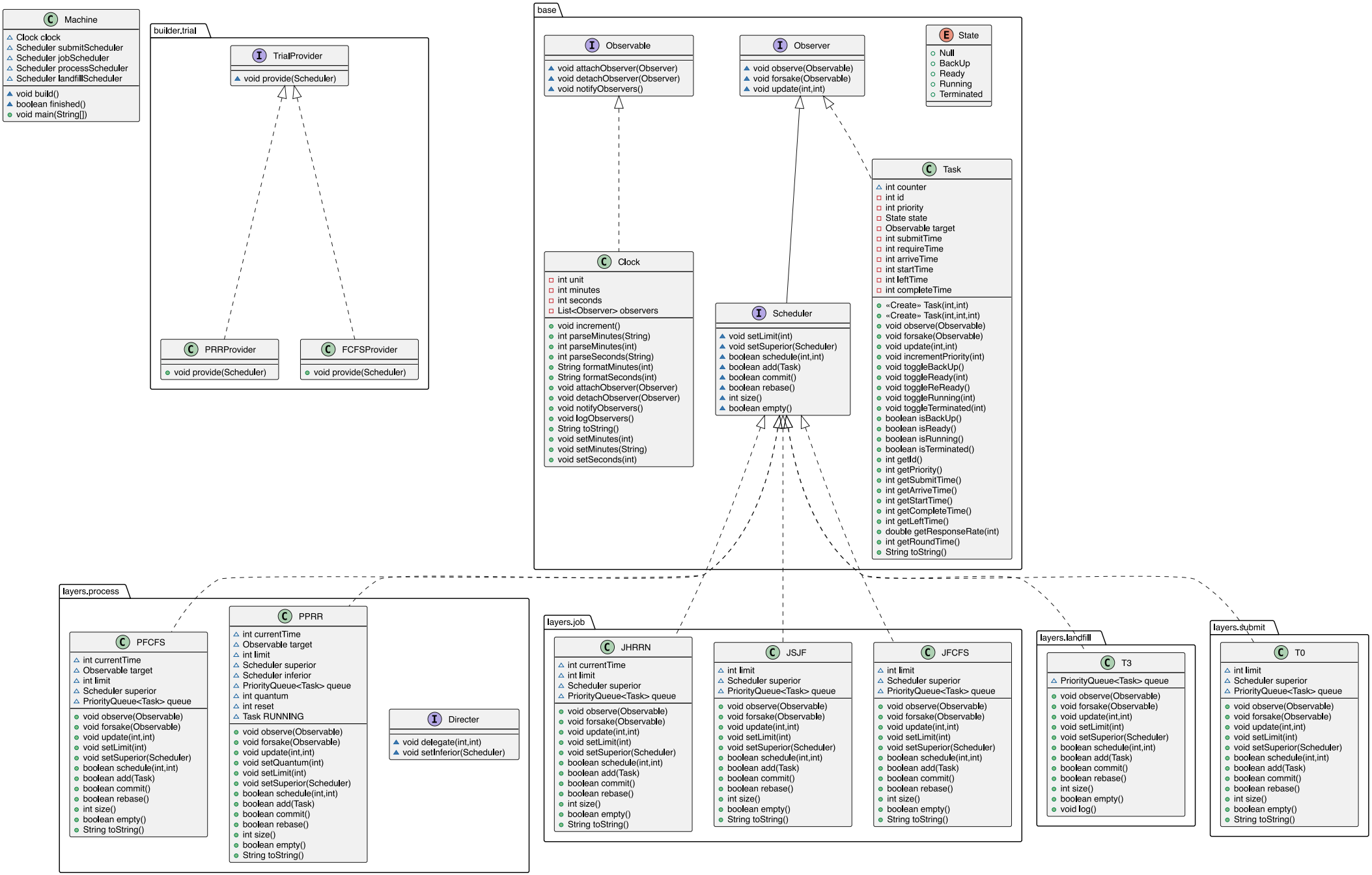


图 1: UML 类图（第一版）


Machine
Clock clock
Scheduler submitScheduler
Scheduler jobScheduler
Scheduler processScheduler
Scheduler landfillScheduler
void build()
boolean finished()
void main(String[])
builder.trial
TrialProvider
void provide(Scheduler)
PRRProvider
void provide(Scheduler)
FCFSProvider
void provide(Scheduler)
base
Observable
void attachObserver(Observer)
void detachObserver(Observer)
void notifyObservers()
Observer
void observe(Observable)
void forsake(Observable)
void update(int,int)
State
Null
BackUp
Ready
Running
Terminated
Clock
int unit
int minutes
int seconds
List<Observer> observers
void increment()
int parseMinutes(String)
int parseMinutes(int)
int parseSeconds(String)
String formatMinutes(int)
String formatSeconds(int)
void attachObserver(Observer)
void detachObserver(Observer)
void notifyObservers()
void logObservers()
String toString()
void setMinutes(int)
void setMinutes(String)
void setSeconds(int)
Scheduler
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
Task
int counter
int id
int priority
State state
Observable target
int submitTime
int requireTime
int arriveTime
int startTime
int leftTime
int completeTime
«Create» Task(int,int)
«Create» Task(int,int,int)
void observe(Observable)
void forsake(Observable)
void update(int,int)
void incrementPriority(int)
void toggleBackUp()
void toggleReady(int)
void toggleReReady()
void toggleRunning(int)
void toggleTerminated(int)
boolean isBackUp()
boolean isReady()
boolean isRunning()
boolean isTerminated()
int getId()
int getPriority()
int getSubmitTime()
int getArriveTime()
int getStartTime()
int getCompleteTime()
int getLeftTime()
double getResponseRate(int)
int getRoundTime()
String toString()
layers.process
PFCFS
int currentTime
Observable target
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
PPRR
int currentTime
Observable target
int limit
Scheduler superior
Scheduler inferior
PriorityQueue<Task> queue
int quantum
int reset
Task RUNNING
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setQuantum(int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
Directer
void delegate(int,int)
void setInferior(Scheduler)
layers.job
JHRRN
int currentTime
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
JSJF
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
JFCFS
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()
layers.landfill
T3
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
void log()
layers.submit
T0
int limit
Scheduler superior
PriorityQueue<Task> queue
void observe(Observable)
void forsake(Observable)
void update(int,int)
void setLimit(int)
void setSuperior(Scheduler)
boolean schedule(int,int)
boolean add(Task)
boolean commit()
boolean rebase()
int size()
boolean empty()
String toString()

2.4 质量评审

基于 2.3 实现的系统，将存在一个致命缺陷：延时。之所以会延时，主要是因为各个时间观察者对于时间流逝的感知顺序是固定、单向的。

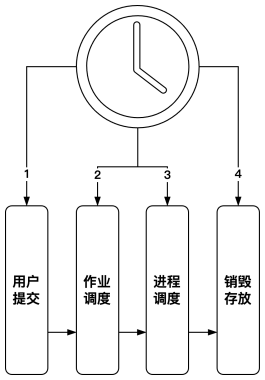


图 2: 通知顺序

虽然从层级上说，作业调度高于（先于）进程调度，但是从顺序上说，在单位时间流逝后，有可能出现先作业调度、再进程调度、再作业调度的情况。

表 3: 错误案例（单道批处理系统）

时刻	收容队列	就绪队列	备注
09:59	{}	{[运行，余 1 分钟]}	初始状态
10:00	{}	{[运行，余 1 分钟]}	用户提交新作业
10:00	{[收容，余 1 分钟]}	{[运行，余 1 分钟]}	内存已满 | 无作业调度
10:00	{[收容，余 1 分钟]}	{[运行，余 1 分钟]}	正在运行 | 无进程调度
10:00	{[收容，余 1 分钟]}	{[结束，余 0 分钟]}	进程自主修改剩余时间
10:01	{[收容，余 1 分钟]}	{[结束，余 0 分钟]}	内存已满 | 无作业调度
10:01	{[收容，余 1 分钟]}	{}	释放进程（进程调度）
10:02	{}	{[就绪，余 1 分钟]}	掉入内存（作业调度）
10:02	{}	{[运行，余 1 分钟]}	运行进程（进程调度）
10:02	{}	{[结束，余 0 分钟]}	进程自主修改剩余时间
...	...	...	...

本错误案例中，10:00～10:01 处理机竟然处于空闲状态。这是因为：

- 进程运行不受进程调度控制，进程调度没有及时将结束进程向上提交；
- 进程调度即使向上提交，作业调度也无法预知未来，并为其创建进程。

下面提出两种可能的改进方案，并加以分析：

调整时间单位 承认并接受延时的存在，并将时间继续细分。例如，倘若单位设定为“秒”，那么在“分钟”维度上看，延时仿佛并不存在。不过，只要没有无限细分，延时累积最终必将影响结果。

优化通信机制 扭转各层对时间流逝固定、单向的感知顺序。拟人地说，各层消息传递机制不能只包含“向上告知”，还要包含“向下问责”。进程调度结束后，倘若可以通知作业调度：“你那里还有什么要我签字处理的吗？”；那么，延时问题说不定就可以迎刃而解。

3 系统实现

3.1 核心实现

在上一节的基础上，添加以下前提假设，以便系统实现：时钟的最小颗粒单位为分钟，所有作业均在 23:59 前提交并结束。《任务书》设计描述中，单位并不统一，如，“进程到达时间（单位时间）”、“作业提交时间（时钟时刻）”、“预计运行时间（小时）”。所以需要在正式输入核心部分前，需要进行数据预处理和单位标准化。

以“多级队列反馈队列”为例，我们需要实现（类图见下页）：

- 核心组件：
  - 任务、状态
  - 时钟（观察对象）
  - 提交“调度”、作业调度、进程调度、销毁“调度”（严格意义上说，提交和销毁不属于调度，它们是对定时提交和进程销毁的一种模拟）
- 边缘组件：
  - 处理机工厂，用不同算法构建各级调度
  - 测试样例工厂，创建任务并添加至提交“调度”

多级调度各司其职，共同作为批处理系统的重要组成部分发挥作用：

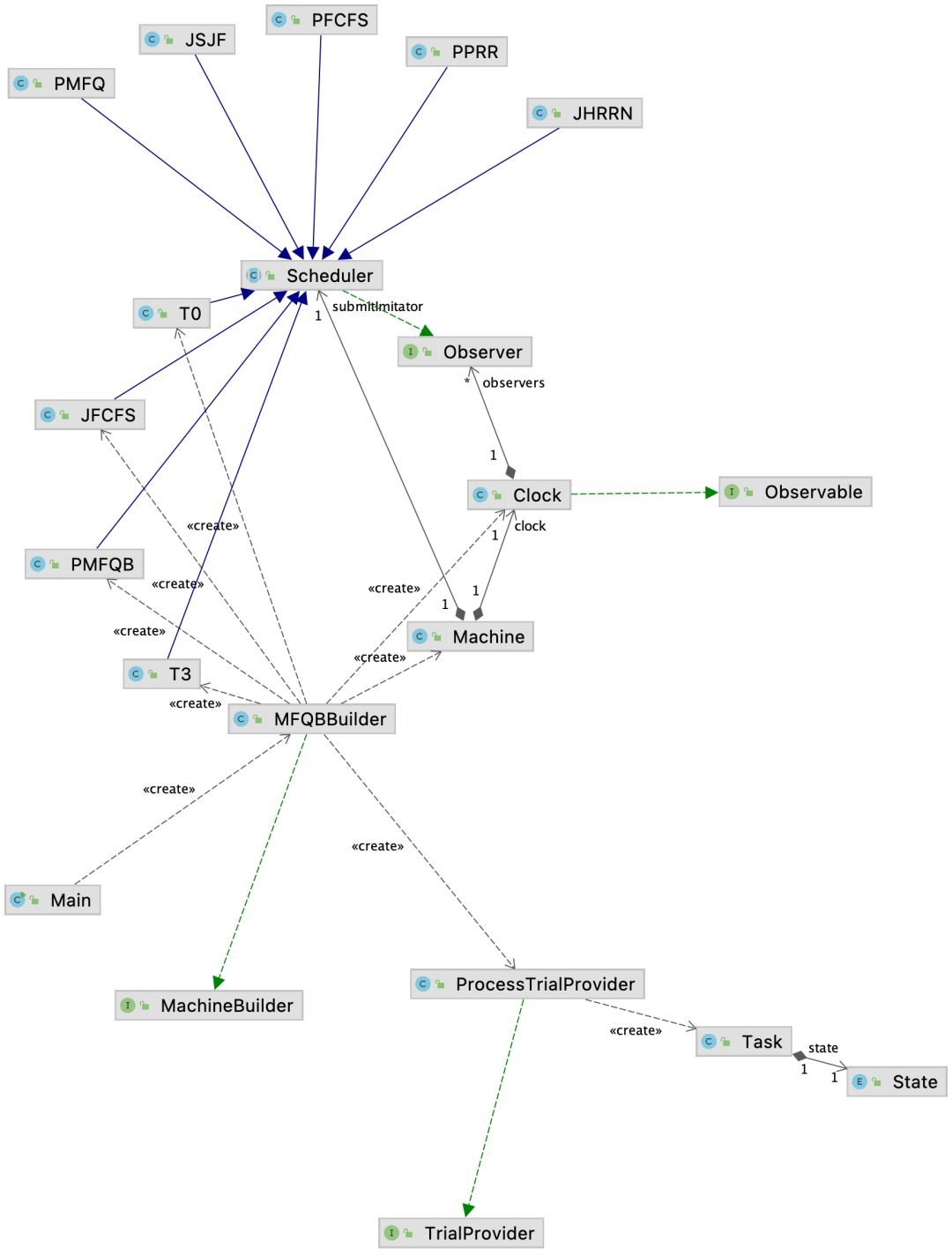

PFCFS
JSJF
PPRR
PMFQ
JHRRN
Scheduler
T0
submitImitator
1
Observer
*
observers
JFCFS
1
Clock
Observable
«create»
1
clock
PMFQB
«create»
«create»
1
1
«create»
Machine
«create»
T3
«create»
MFQBBuilder
«create»
«create»
Main
MachineBuilder
ProcessTrialProvider
«create»
Task
state
1
1
State
TrialProvider

图 3: UML 类图（第二版）

表 4: 常用的处理机调度策略

多级调度	调度方式	调度时机	调度次序	任务状态
用户提交	↑	提交时刻 ≤ 当前时刻	提交时间	未提交
作业调度	↑	时间流逝 & 上层命令	*	收容
进程调度	↑、↔、↓	时间流逝	**	就绪、运行
进程销毁				销毁

[1] ↑ 移交上层调度，↔ 本层级调度，↓、委托下层调度
[2] * 提交时间、剩余时间、响应比
[3] ** 优先级（多级队列优先次序）

3.1.1 作业调度

实现单道批处理系统，作业调度分别使用“先来先服务算法”、“短作业优先算法”、“高响应比优先算法”，进程调度统一使用“先来先服务算法”。

图 4: 作业调度层

作业调度的职责是按照一定的策略，向进程调度提交任务（填入内存）。FCFS、HRRN、SJF 的不同主要体现于调度次序，即优先队列的比较方式。本节主要以高响应比优先实现为例，介绍抽象调度类与作业调度层。

```java
public abstract class Scheduler implements Observer {
    protected int LIMIT = Integer.MAX_VALUE;
    protected Scheduler superior = null;
    protected Scheduler inferior = null;

    public void setLIMIT(int LIMIT) {this.LIMIT = LIMIT;}

    public void setSuperior(Scheduler superior) {this.superior = superior;}

    public void setInferior(Scheduler inferior) {this.inferior = inferior;}

    @Override
    public void observe(Observable target) {target.attachObserver(this);}

    @Override
    public void forsake(Observable target) {target.detachObserver(this);}

    public abstract boolean add(Task t);

    public abstract boolean reportToSuperior();

    public abstract boolean notifyToInferior();

    public abstract boolean rebase();

    public abstract int size();

    public abstract boolean isEmpty();

    public abstract void shuffle();
}
```

下面对于各个接口，分别解释具体实现：

Listing 1: 收容队列（基于响应比的优先队列）

```
private PriorityQueue<Task> queue = new PriorityQueue
    <>(new Comparator<Task>() {
     @Override
     public int compare(Task o1, Task o2) {
         return (o2.getResponseRate(Clock.minutes) - o1
             .getResponseRate(Clock.minutes) > 0) ? (1)
             : (-1);
     }
});
```

Listing 2: 调度方式（向上提交），调度时机（时间流逝）

```
@Override
public void update(int currentTime, int elapseUnit) {
    boolean flag = true;
    while (queue.peek() != null && flag) {
        flag = reportToSuperior();
    }
}
```

Listing 3: 添加作业（收容状态）

```
@Override
public boolean add(Task t) {
    if (queue.size() >= LIMIT)
        return false;

    t.toggleBackup();
    queue.offer(t);
    return true;
}
```

Listing 4: 模拟为作业创建进程，装填进入内存

```java
@Override
public boolean reportToSuperior() {
    if (superior == null || queue.isEmpty())
        return false;

    shuffle();
    if (!superior.add(queue.peek()))
        return false;

    return rebase();
}
```

Listing 5: 在本层队列中移除任务

```java
@Override
public boolean rebase() {
    if (queue.isEmpty())
        return false;

    queue.poll();
    return true;
}
```

Listing 6: 响应比动态变化，刷新队列顺序

```java
@Override
public void shuffle() {
    if (queue.size() >= 2) {
        queue.offer(queue.poll());
    }
}
```

需要特别说明的是，短作业优先（SJF, Shortest Job First）也有抢占式的版本，通常被称为最短剩余时间优先。即，每当就绪队列改变时都需要进行调度，如果新到达的进程剩余时间比当前运行进程剩余时间更短，则由新进程抢占处理机，当前运行进程重新回到就绪队列。

3.1.2 进程调度

实现多道批处理系统（容量无限），作业调度统一使用“先来先服务”，进程调度分别使用“基于动态优先级的时间片轮转法”、“多级反馈队列法”。

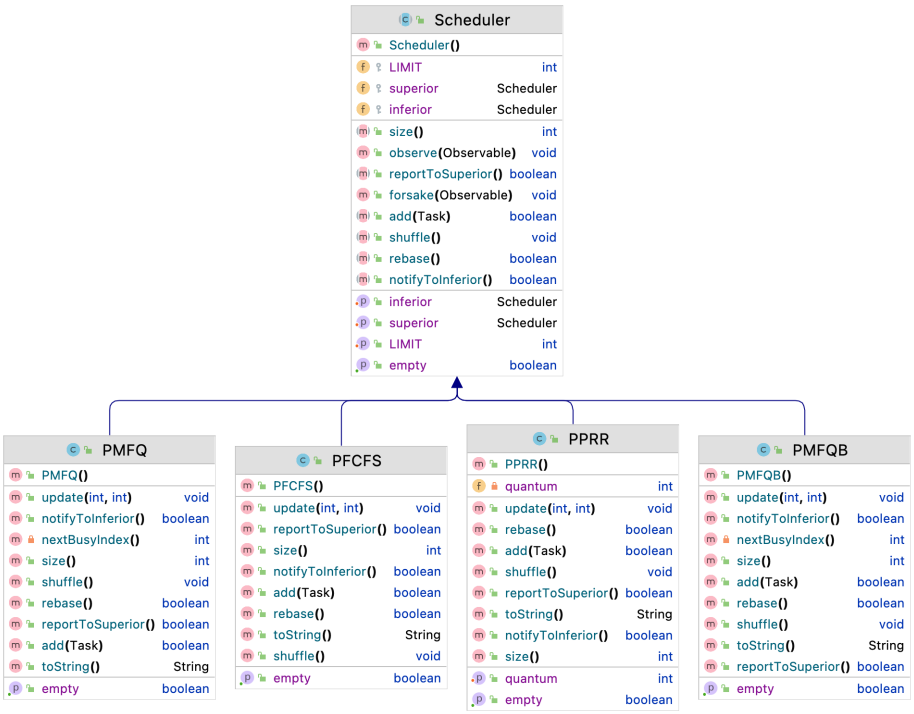


图 5: 进程调度层

本节主要介绍四种算法的实现异同。共性，例如，只有进程调度会通知下层。

Listing 7: 命令下层再次进行调度

```
1 @Override
2 public boolean notifyToInferior() {
3     if (inferior == null)
4         return false;
5
6     return inferior.reportToSuperior();
7 }
```

此外，进程调度的大体框架如下：

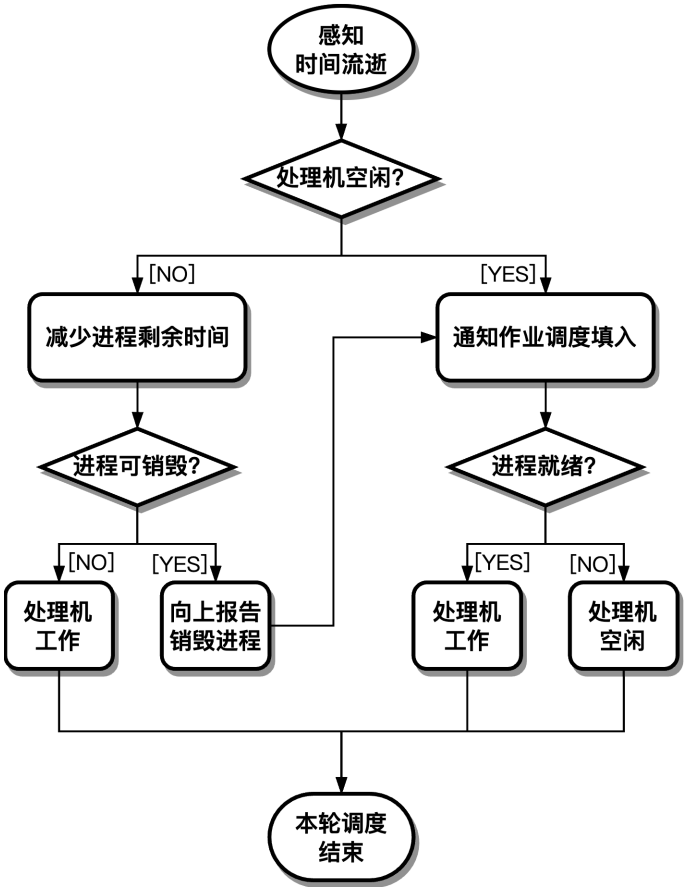


图 6: 进程调度（以先来先服务为例）

不过，不同算法实现细节有所差别，下面结合代码进行分析。这里采用改进通信机制的方式避免延时问题，即，进程调度后，若向上提交销毁进程导致内存存在空余，则通知作业调度并允许其提交新的进程。

Listing 8: 先来先服务

```
1 private boolean leisure = true;
2 private PriorityQueue<Task> queue = new PriorityQueue
      <>(Comparator.comparingInt(Task::
      getProcessArriveTime));
3
```

```
4  @Override
5  public void update(int currentTime, int elapseUnit) {
6      shuffle();
7
8      // existing a running process
9      if (!leisure) {
10         queue.peek().decrementLeftTime(currentTime,
               elapseUnit);
11         if (queue.peek().isTerminated()) {
12             reportToSuperior();
13             leisure = true;
14         } else {
15             leisure = false;
16         }
17     }
18
19     // needy for a ready process
20     if (leisure) {
21         notifyToInferior();
22         if (queue.peek() != null) {
23             queue.peek().toggleRunning(currentTime);
24             leisure = false;
25         }
26     }
27 }
```

基于动态优先级的时间片轮转要求对时间片动态判定修改。此外，程序没有记录运行进程的引用，所以必须通过 tricks 保证运行进程优先级永远最高。

Listing 9: Getter（以优先级为例）

```
1  public int getPriority() {
2      int factor = (isRunning()) ? (-1) : (1);
3      return priority * factor;
4  }
```

Listing 10: 基于动态优先级的时间片轮转

```
1  private boolean leisure = true;
2  private int quantum = 2;
3  private int dynamicTimeSlice = quantum;
4  private PriorityQueue<Task> queue = new PriorityQueue
       <>(new Comparator<Task>() {
5      @Override
6      public int compare(Task o1, Task o2) {
7          return (o1.getPriority() - o2.getPriority() !=
                   0) ? (o1.getPriority() - o2.getPriority())
                   : (o1.getJobSubmitTime() - o2.
                  getJobSubmitTime());
8          // return o1.getPriority() - o2.getPriority();
9      }
10 });
11
12 public void setQuantum(int quantum) {
13     this.quantum = quantum;
14 }
15
16 public void update(int currentTime, int elapseUnit) {
17     shuffle();
18
19     // existing a running process within the last
          minute
20     if (!leisure) {
21         queue.peek().decrementLeftTime(currentTime,
              elapseUnit);
22         dynamicTimeSlice -= elapseUnit;
23         // terminated
24         if (queue.peek().isTerminated()) {
25             queue.peek().incrementPriority(2);
26             reportToSuperior();
```

```
27              // reset time slice
28              leisure = true;
29          } else if (queue.peek().isRunning()) {
30              // brute-force offline
31              if (dynamicTimeSlice == 0) {
32                  queue.peek().incrementPriority(2);
33                  queue.peek().toggleReReady();
34                  shuffle();
35                  leisure = true;
36              }
37              // slice is still adequate for continuous
                    running
38              else {
39                  leisure = false;
40              }
41          }
42      }
43
44      // needy for a ready process
45      if (leisure) {
46          dynamicTimeSlice = quantum;
47          notifyToInferior();
48          if (queue.peek() != null) {
49              queue.peek().toggleRunning(currentTime);
50              leisure = false;
51          }
52      }
53
54  }
```

多级反馈队列法：设置多级就绪队列，各级队列优先级从高到低，时间片由小到大。新进程到达时，先进入第一级队列，按“先来先服务”原则排队等待。若时间片用完进程仍未结束，则将该进程移入下级队列队尾（直到最后一级）。只有第 k 级队列为空，才会为 $k+1$ 级队头进程分配时间片。

Listing 11: 多级反馈队列（数据结构）

```
private List<Deque<Task>> queues = new ArrayList<>();
private boolean leisure = true;
private int busyIndex = 0;
private int quantum = 1;
private int dynamicTimeSlice = quantum;

public PMFQ() {
    Deque<Task> top = new ArrayDeque<>();
    queues.add(top);
    Deque<Task> mid = new ArrayDeque<>();
    queues.add(mid);
    Deque<Task> btm = new ArrayDeque<>();
    queues.add(btm);
```

多级反馈队列法同样可分为抢占式和非抢占式，这两种实现的主要区别在于“调度时机”。假设当前运行进程优先级并非顺位第一，即使时间片未运行完，抢占式调度也会收回剩余时间片使优先级更高的新进程获得处理机资源，并将被抢占处理机的进程重新放回原队列队尾。而非抢占式调度须要等时间片轮转才可以调度。

Listing 12: 多级反馈队列（非抢占式调度）

```
@Override
public void update(int currentTime, int elapseUnit) {
    // existing a running process within the last
       minute
    if (!leisure) {
        Deque<Task> queue = queues.get(busyIndex);
        queue.peek().decrementLeftTime(currentTime,
            elapseUnit);
        dynamicTimeSlice -= elapseUnit;
        // terminated
        if (queue.peek().isTerminated()) {
            reportToSuperior();
```

```
11              // reset time slice
12              leisure = true;
13          } else if (queue.peek().isRunning()) {
14              // brute-force offline
15              if (dynamicTimeSlice == 0) {
16                  queue.peek().toggleReReady();
17                  int nextIndex = Math.min(busyIndex +
                        1, 2);
18                  // next level or bottom
19                  queues.get(nextIndex).offer(queue.poll
                        ());
20                  leisure = true;
21              }
22              // slice is still adequate for continuous
                    running
23              else {
24                  leisure = false;
25              }
26          }
27      }
28
29      // needy for a ready process
30      if (leisure) {
31          notifyToInferior();
32          // somebody ready
33          if (nextBusyIndex() != -1) {
34              busyIndex = nextBusyIndex();
35              quantum = (int) Math.pow(2, busyIndex);
36              dynamicTimeSlice = quantum;
37
38              Deque<Task> queue = queues.get(busyIndex);
39              queue.peek().toggleRunning(currentTime);
40              leisure = false;
```

```
41          }
42      }
43  }
```

Listing 13: 多级反馈队列（抢占式调度）

```
1   @Override
2   public void update(int currentTime, int elapseUnit) {
3       // existing a running process within the last
            minute
4       if (!leisure) {
5           Deque<Task> queue = queues.get(busyIndex);
6           queue.peek().decrementLeftTime(currentTime,
                elapseUnit);
7           dynamicTimeSlice -= elapseUnit;
8           // terminated
9           if (queue.peek().isTerminated()) {
10              reportToSuperior();
11              // reset time slice
12              leisure = true;
13          } else if (queue.peek().isRunning()) {
14              // brute-force offline
15              if (dynamicTimeSlice == 0) {
16                  queue.peek().toggleReReady();
17                  int nextIndex = Math.min(busyIndex +
                        1, 2);
18                  // next level or bottom
19                  queues.get(nextIndex).offer(queue.poll
                        ());
20                  leisure = true;
21              }
22              // slice is still adequate for continuous
                    running
23              else {
```

```
24                  // nobody can forcibly occupy
25                  if (nextBusyIndex() == busyIndex) {
26                      leisure = false;
27                  } else {
28                      // FIXME
29                      queue.peek().toggleReReady();
30                      queue.offer(queue.poll());
31                      leisure = true;
32                  }
33              }
34          }
35      }
36
37      // needy for a ready process
38      if (leisure) {
39          notifyToInferior();
40          // somebody ready
41          if (nextBusyIndex() != -1) {
42              busyIndex = nextBusyIndex();
43              quantum = (int) Math.pow(2, busyIndex);
44              dynamicTimeSlice = quantum;
45
46              Deque<Task> queue = queues.get(busyIndex);
47              queue.peek().toggleRunning(currentTime);
48              leisure = false;
49          }
50      }
51  }
```

3.2 边缘实现

选择使用 Android 移动平台部署应用，以省去云服务器环境搭建。

3.2.1 依赖配置

```
android {
    namespace 'edu.wust.durui'
    compileSdk 32
    defaultConfig {
        vectorDrawables.useSupportLibrary = true
        multiDexEnabled true
        applicationId "edu.wust.durui"
        minSdk 29
        targetSdk 32
        versionCode 1
        versionName "1.0"
        ...
    }
    ...
    viewBinding {
        enabled = true
    }
    dataBinding {
        enabled = true
    }
}
dependencies {
    ...
    implementation 'com.google.android.material:material:1.6.1'
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.4.1"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.5.1"
    implementation "androidx.multidex:multidex:2.0.1"
    implementation "androidx.fragment:fragment:1.5.5"
    implementation 'com.github.blackfizz:eazegraph:1.2.2@aar'
    implementation 'com.nineoldandroids:library:2.4.0'
    implementation 'com.github.vipulasri:timelineview:1.1.5'
}
```

3.2.2 核心技术

MVVM 应用架构定义了应用的各个部分之间的界限以及每个部分应承担的职责，遵循“关注点分离”、“数据驱动界面”、“单一数据源和单向数据流”，避免在 Activity 或 Fragment 编写所有代码。

Fragment Fragment 组件介于 Activity 和 View：一方面，它未继承 Context, 无法独立存在，必须作为宿主视图的一部分；可另一方面，它又可以定义管理自己的布局，具有自己的生命周期（Lifecycle）。

ViewModel 配置更改（例如旋转屏幕）时，UI 数据可能丢失，是因为界面被系统自动销毁重绘（和会输销毁整个程序不同），同时，界面类的成员变量也被回收。一个解决方案是，将数据的所有权和界面交互逻辑分离开；如此，系统只回收 Activity 类中的对数据类的“引用”，并不会影响数据类本身。

当 ViewModel 中存在一个 LiveData 容器，可以指定当前数据的观察者（Observer），即每当 LiveData 中存放的数据发生变化时，观察者可以全自动地执行操作，例如，改变 UI 界面中的数据。相比起定时检测更新和通知模式，观察者具有强大生命力和无比优越性。

RecyclerView 核心视图组件之一。RecyclerView 可以轻松高效地显示大量数据，当提供数据并定义每个列表项的外观后，RecyclerView 库会根据需要动态创建元素。

DatePicker 核心视图组件之一。DatePicker 继承自 FrameLayout 类，日期选择控件的主要功能是向用户提供包含年、月、日的日期数据并允许用户对其修改。如果要捕获用户修改日期选择控件中的数据事件，需要为 DatePicker 添加 OnDateChangedListener 监听器。

EazeGraph 核心视图组件之一。轻量级的图表库[4]，支持生成四种图表：条形图，层叠柱状图，饼状图，折线图。

TimelineView 核心视图组件之一。轻量级时间视图库[5]，支持呈现时间线，应用广泛，如物流追踪历史节点等。

[4]https://github.com/paulroehr/EazeGraph

[5]https://github.com/vipulasri/Timeline-View

3.3 界面设计

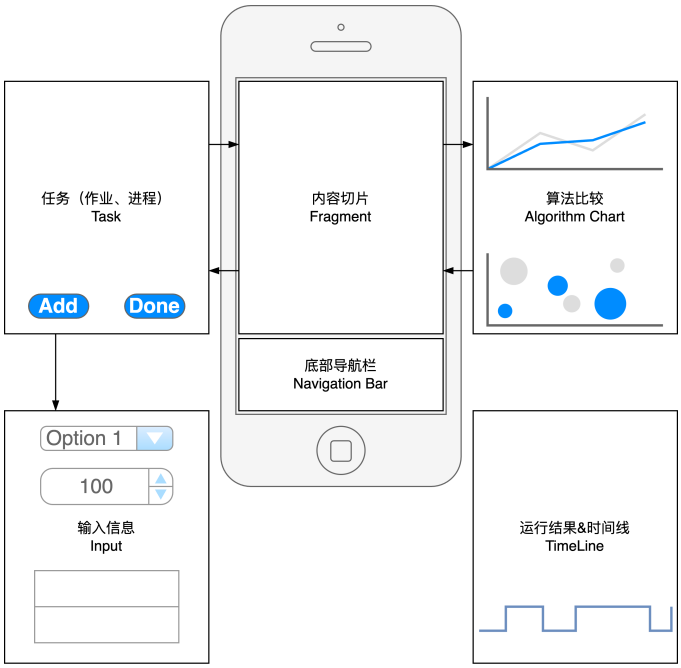


图 7: 界面原型设计图

4 系统测试

4.1 测试用例

表 5: 测试用例（先来先服务，平均周转时间 1.725h）

作业	进入时刻	运行时间（分钟）	开始时刻	完成时刻	周转时间（分钟）
1	08:00	120	08:00	10:00	120
2	08:30	30	10:00	10:30	120
3	09:00	6	10:30	10:36	96
4	09:30	12	10:36	10:48	78

表 6: 测试用例（短作业优先，平均周转时间 1.550h）

作业	进入时刻	运行时间（小时）	开始时刻	完成时刻	周转时间（小时）
1	08:00	2.00	08:00	10:00	2.00
2	08:30	0.50	10:18	10:48	2.30
3	09:00	0.10	10:00	10:06	1.10
4	09:30	0.20	10:06	10:18	0.80

表 7: 测试用例（高响应比优先，平均周转时间 1.625h）

作业	进入时刻	运行时间（小时）	开始时刻	完成时刻	周转时间（小时）
1	08:00	2.00	08:00	10:00	2.00
2	08:30	0.50	10:06	10:36	2.10
3	09:00	0.10	10:00	10:06	1.10
4	09:30	0.20	10:36	10:48	1.30

表 8: 测试用例（248 抢占式多级反馈队列）

作业	进入时刻	服务时间	完成时刻	周转时间
A	0	7	11	11
B	5	4	19	14
C	7	13	30	23
D	12	9	33	21

表 9: 测试用例（124 非抢占式多级反馈队列）

作业	进入时刻	服务时间	完成时刻	周转时间
A	0	3	3	3
B	2	6	17	15
C	4	3	18	14
D	6	5	20	14
E	8	2	14	6

4.2 运行结果

见尾页。

5 设计总结

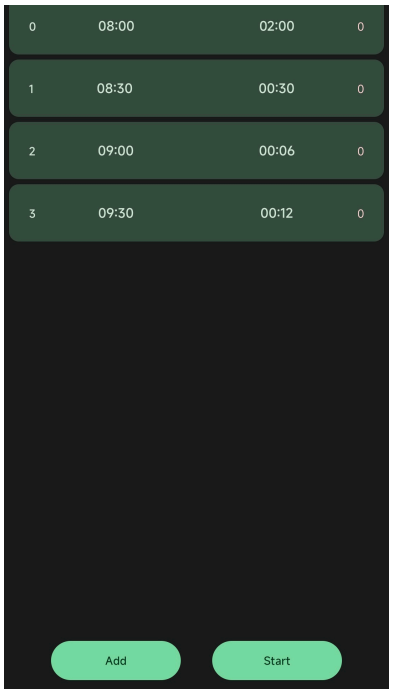
宏观来说，本次课程设计，再一次使我体验了结构化编程、面向对象程序设计的魅力。

- 自顶向下，逐步求精。例如，本程序模拟处理机，接受用户输入作业序列，输出调度过程及周转时间。处理机的模拟实现可以拆分为对时钟，调度，任务的实现。调度的模拟实现可以分拆为用户提交、作业调度、进程调度、进程销毁。
- 抽象、封装、多态、继承。例如，各级调度虽然各司其职，分工合作，但是又有相类似行为：观察时间流逝、接受任务、移除任务、顺位排序、向上级调度提交任务等。正因此，本系统极大地支持扩展新的调度算法，并通过分级的形式相互排列组合。

微观地说，本次课程设计，使我有机会深入了解操作系统处理机调度的种种细节，并有机会尝试移动平台软件设计（基于 Java 的 Android）的最佳实践。

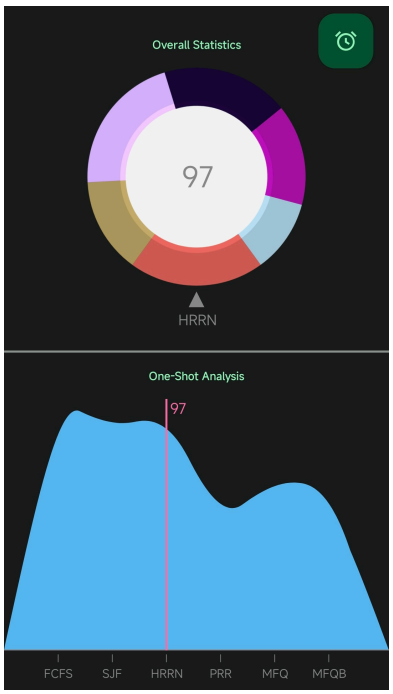
- 例如，以 Activity 作为 Fragment 的宿主，结合 ViewModel（LiveData）时，需要注意 ViewModelStoreOwner 统一，否则无法在切片间共享数据模型。
- 又如，PriorityQueue 结合 Comparator 使用，动态优先级 getter/setter 的 tricks，深拷贝与克隆接口实现。
- 再如，多级反馈队列分为抢占时间片和不抢占时间片两种版本，两种的实现方式类似，唯一的区别在于更新工作队列索引的时机，所谓的“抢占”事实上是每单位时间判断（切换）一次，而所谓的“非抢占”需处理机处于“空闲”状态判断（切换）一次。

虽然本系统有一定的健壮性，可以抵御一些异常输出，但是依然存在不足，有待改进：如不支持输入多级队列的数量（大于三个）及自定义时间片（非指数型增长），由于屏幕限制时间线界面交互设计有待改进。

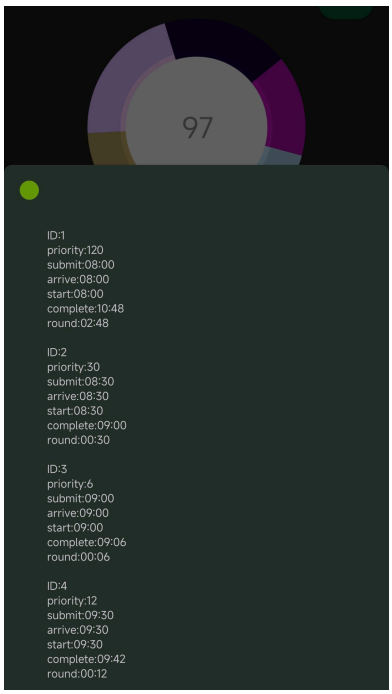


(a) 任务列表

(b) 时刻输入



(c) 性能比较



(d) 时间线

图 8: 运行截图