操作系统实验报告

学院：泰山学堂

姓名：苏柏瑞

学号：201600301108

指导老师：杨兴强

助教：程鲁豫

同组者：周玉昆

实验目的

获取知识

> 了解Liunx系统的原理，进而对操作系统有更多的认识。

提升能力

> - 读以及分析原代码的能力
> - 编程能力
> - 计算机系统能力
> - 沟通协作能力
> - 表达能力

实验内容

目标

> Linux操作系统运行过程可视化。

要求

> 着重可视化一个模块，需要把代码完全理解（闭上眼睛能把代码重述一次）。

实际情况

> 我们小组经过讨论，确定可视化进程模块。我是负责可视化部分，周玉昆是负责获取数据部分。

实验环境

读源码工具

> - Sublime Text
> - Linux 0.11 完全解读

获得数据工具

> - Linux虚拟机
> - 2015级王天浩学长的环境

可视化工具

> - python处理数据。
> - processing可视化数据。

实验步骤

读源代码

分析代码的体会

- Linus成功的因素让我很有启发：
  - 外在原因：
    - unix：linux以这个为原型，但这个不是开源的。
    - minix：克隆unix，并且开放源码。
    - GNU：linux只有内核，但像bash shell这些软件还是该计划提供的。
    - posix：朝着正规路上发展。
    - internet：遍布世界的计算机人才一起贡献。
  - 内在原因：
    - 喜欢探索：喜欢鼓捣计算机，测试计算机的性能和极限。
    - 站在巨人肩上成功：认真学习了intel的硬件知识，和minix系统，知道了其好的地方和坏的地方。
    - 会分享：将自己的成果分享到网上，并且请大家来一直完善。
- 复杂的工程都是从一个简单的main函数开始，所以很多东西并没想象中的那么复杂。
- 所有的进程都是由0号进程产生的，最后又会只剩下0号进程。有种“一生二，二生三，三生万物，万物归一”的感觉。
- Linus的代码让人看上去赏析悦目，有两点很重要:
  - 函数和变量的命名命很恰当，让人一看就知道有什么作用。
  - 逻辑很清晰。

重点分析的代码描述

> main.c：
>
> - 主要的功能：创建第一个进程。
> - 发现：不管操作系统多么的复杂，它都是从main函数作为入口的。所以main.c里面可以算是整个操作系统最核心的代码。

创建第一进程

```
void main (void)        /* This really IS void, no error here. */
{
//....
  // 下面过程通过在堆栈中设置的参数，利用中断返回指令切换到任务0。
  move_to_user_mode ();     // 移到用户模式。 （include/asm/system.h，第1
 行）
  if (!fork ())
    {               /* we count on this going ok */
      init ();
    }
  /*
   * NOTE!! For any other task 'pause()' would mean we have to get a
   * signal to awaken, but task0 is the sole exception (see 'schedule()')
   * as task 0 gets activated at every idle moment (when no other tasks
   * can run). For task0 'pause()' just means we go check if some other
   * task can run, and if not we return here.
   */
  /* 注意!! 对于任何其它的任务，'pause()'将意味着我们必须等待收到一个信号才会返
   * 回就绪运行态，但任务0（task0）是唯一的意外情况（参见'schedule()'），因为任务0
 在
   * 任何空闲时间里都会被激活（当没有其它任务在运行时），因此对于任务0'pause()'仅意味
着
   * 我们返回来查看是否有其它任务可以运行，如果没有的话我们就回到这里，一直循环执行'pa
use()'。
   */
  for (;;)
    pause ();
}
```

init函数（感兴趣的部分）

```
void init (void)
{
  int pid, i;
    //....
  if (!(pid = fork ()))
    {
      close (0);
      if (open ("/etc/rc", O_RDONLY, 0))
```

```
    _exit (1);      // 如果打开文件失败，则退出(/lib/_exit.c,10)。
      execve ("/bin/sh", argv_rc, envp_rc); // 装入/bin/sh 程序并执行。
      _exit (2);        // 若execve()执行失败则退出(出错码2,“文件或目录不存
在”)。
    }
  // 下面是父进程执行的语句。wait()是等待子进程停止或终止，其返回值应是子进程的进程号
(pid)。
  // 这三句的作用是父进程等待子进程的结束。&i 是存放返回状态信息的位置。如果wait()返
回值不
  // 等于子进程号，则继续等待。
  if (pid > 0)
    while (pid != wait (&i))
  //...
  _exit (0);            /* NOTE! _exit, not exit() */
}
```

> sched.h：
>
> - 功能：和进程相关的一些数据结构和函数。
> - 发现：描述一个进程其实很简单。
>   - 进程所处的状态：state，counter，priority
>   - 进程的在全局的地位：pid，father
>   - 进程的局部信息：start_code，end_code，end_data，brk，star_code

进程的结构（感兴趣的内容）

```
struct task_struct
{
/* these are hardcoded - don't touch */
  long state;   //任务的运行状态（-1 不可运行，0 可运行(就绪)，>0 已停止）。

  long counter; //任务运行时间计数(递减)（滴答数），运行时间片。
  long priority; //运行优先数。任务开始运行时counter = priority，越大运行越长。
  //....
/* various fields */
  //...
  unsigned long start_code, //代码段地址
      end_code, //代码长度（字节数）
      end_data, //代码长度 + 数据长度（字节数）
      brk, //总长度（字节数）
      start_stack;//堆栈段地址
  long pid, //进程标识号(进程号)。
       father; //父进程号。
  //...
};
```

存储进程的数据结构

```
#define FIRST_TASK task[0]  // 任务0 比较特殊，所以特意给它单独定义一个符号。
#define LAST_TASK task[NR_TASKS-1]  // 任务数组中的最后一项任务。
```

调度函数声明

```
extern void sched_init (void);
// 进程调度函数。( kernel/sched.c, 104 )
extern void schedule (void);
// 异常(陷阱)中断处理初始化函数，设置中断调用门并允许中断请求信号。( kernel/traps.c,
 181 )
```

> sched.c
>
> - 功能：和进程调度相关的算法。
> - 发现：通过轮询的方式，选择第一个比某一个固定值大的counter的进程。
>   - 每一个priority都是一样的。
>   - counter是根据priority计算的。

schedule函数

```
void schedule (void)
{
  int i, next, c;
  struct task_struct **p;   // 任务结构指针的指针。
//...
  /* 这里是调度程序的主要部分 */

  while (1){
      c = -1;
      next = 0;
      i = NR_TASKS;
      p = &task[NR_TASKS];
      // 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的数组槽。比
较每个就绪
      // 状态任务的counter（任务运行时间的递减滴答计数）值，哪一个值大，运行时间还不
长，next 就
      // 指向哪个的任务号。
      while (--i) {
      if (!*--p)
        continue;
      if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
        c = (*p)->counter, next = i;
    }
```

```
      // 如果比较得出有counter 值大于0 的结果，则退出124 行开始的循环，执行任务切换（141 行）。
      if (c)
    break;
      // 否则就根据每个任务的优先权值，更新每一个任务的counter 值，然后回到125 行重新比较。
      // counter 值的计算方式为counter = counter /2 + priority。[右边counter=0??]
      for (p = &LAST_TASK; p > &FIRST_TASK; --p)
    if (*p)
      (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
    }
  switch_to (next);     // 切换到任务号为next 的任务，并运行之。
}
```

> fork.c
>
> - 功能：创建新的进程。
> - 发现：采用的是copy on write技术。

复制内存（感兴趣部分）

```
int copy_mem (int nr, struct task_struct *p)
{
  //...
  p->start_code = new_code_base;
  //...
  return 0;
}
```

复制进程（感兴趣的部分）

```
long last_pid = 0;
int copy_process (int nr, long ebp, long edi, long esi, long gs, long none,
          long ebx, long ecx, long edx,
          long fs, long es, long ds,
          long eip, long cs, long eflags, long esp, long ss ){
  struct task_struct *p;
  int i;
  struct file *f;

  p = (struct task_struct *) get_free_page ();  // 为新任务数据结构分配内存。
  if (!p)           // 如果内存分配出错，则返回出错码并退出。
    return -EAGAIN;
  task[nr] = p;         // 将新任务结构指针放入任务数组中。
```

```
// 其中nr 为任务号，由前面find_empty_process()返回。
  *p = *current;        /* NOTE! this doesn't copy the supervisor stack
 */
/* 注意！这样做不会复制超级用户的堆栈 */ （只复制当前进程内容）。
  p->state = TASK_UNINTERRUPTIBLE;  // 将新进程的状态先置为不可中断等待状态。
  p->pid = last_pid;        // 新进程号。由前面调用find_empty_process()得到。
  p->father = current->pid; // 设置父进程号。
  p->counter = p->priority;
 //....
  p->state = TASK_RUNNING;  /* do this last, just in case */
/* 最后再将新任务设置成可运行状态，以防万一 */
  return last_pid;      // 返回新进程号（与任务号是不同的）。
}
```

获取进程号(感兴趣部分)

```
int find_empty_process (void){
  int i;
repeat:
  if ((++last_pid) < 0)
    last_pid = 1;
  for (i = 0; i < NR_TASKS; i++)
    if (task[i] && task[i]->pid == last_pid)
      goto repeat;
  for (i = 1; i < NR_TASKS; i++)    // 任务0 排除在外。
    if (!task[i])
      return i;
  return -EAGAIN;
}
```

> sytem_call.s
>
> - 功能：创建新的进程。
> - 发现：创建新的进程的过程是：
>   - 首先调用C 函数find_empty_process()，取得一个进程号pid。若返回负数则说明目前任务数组已满。
>   - 然后调用copy_process()复制进程。

fork函数的汇编代码

```
1.#### sys_fork()调用，用于创建子进程，是system_call 功能2。原形在include/linux/sys.h
 中。
2.# 首先调用C 函数find_empty_process()，取得一个进程号pid。若返回负数则说明目前任务数组
3.# 已满。然后调用copy_process()复制进程。
4..align 2
5._sys_fork:
```

```
6.call _find_empty_process # 调用find_empty_process()(kernel/fork.c,135)。
7.testl %eax,%eax
8.js 1f
9.push %gs
10.pushl %esi
11.pushl %edi
12.pushl %ebp
13.pushl %eax
14.call _copy_process # 调用C 函数copy_process()(kernel/fork.c,68)。
15.addl $20,%esp # 丢弃这里所有压栈内容。
16.1: ret
```

> exit.c
>
> - 功能：杀死一个进程，并且释放资源。
> - 发现：release是杀死一个进程的最后一步，可以在这个地方加一个断点获得数据。

release函数

```
void release (struct task_struct *p){
  int i;
  if (!p)
    return;
  for (i = 1; i < NR_TASKS; i++)    // 扫描任务数组，寻找指定任务。
    if (task[i] == p){
    task[i] = NULL;      // 置空该任务项并释放相关内存页。
    free_page ((long) p);
    schedule ();         // 重新调度。
    return;
      }
  panic ("trying to release non-existent task");    // 指定任务若不存在则死机。
}
```

系统运行的形式化描述

主要涉及一个进程的生命周期，见后面的讨论部分。

获取数据

- 内核运行数据输出方法：
  - 用2015级王天浩学长的环境。
  - 获取数据由同组者周玉昆完成。
- 周玉昆得到的数据：
  - 文件名字：output.txt

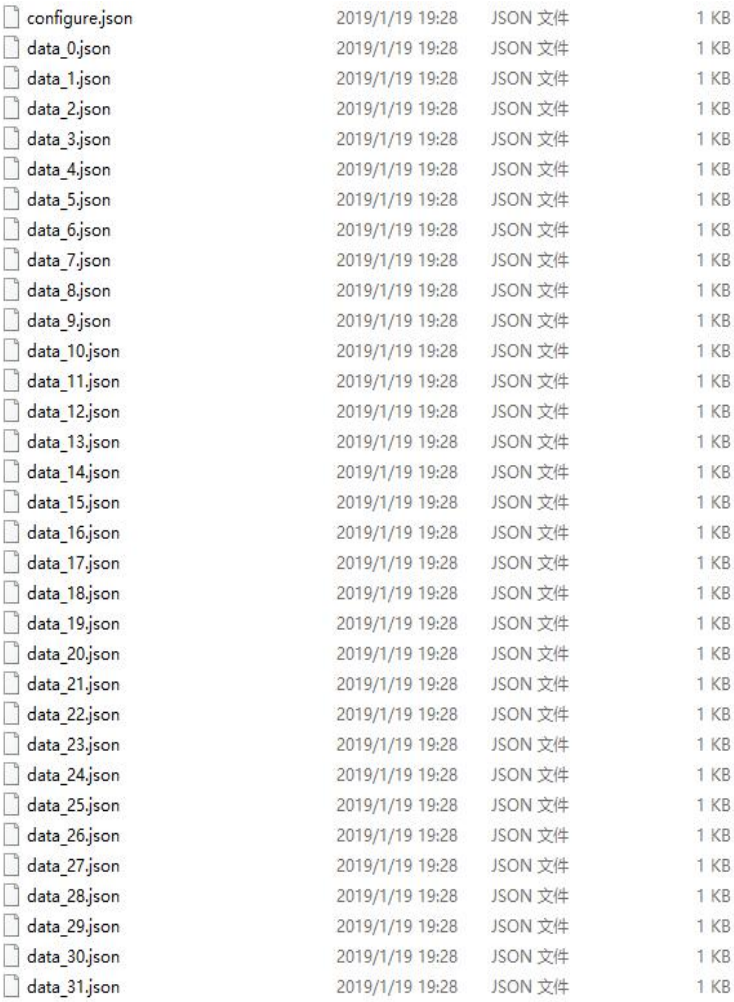
- 内容说明
  - 该文件包含了不同时刻task数组里的进程的对应状态。
  - 不同的时刻用$号隔开，进程和进程之间用#号隔开。

```
1.Operation: 3
2.process No. 0
3.current: 0
4.state: 0
5.pid: 0
6.father: -1
7.counter: 14
8.priority: 15
9.#
10.Operation: 3
11.process No. 1
12.current: 0
13.state: 0
14.pid: 1
15.father: 0
16.counter: 15
17.priority: 15
18.#
19.$
20.
21.Operation: 4
22.process No. 0
23.current: 0
24.state: 1
25.pid: 0
26.father: -1
27.counter: 14
28.priority: 15
29.#
30.Operation: 4
31.process No. 1
32.current: 0
33.state: 0
34.pid: 1
35.father: 0
36.counter: 15
37.priority: 15
38.#
39.$
40.......
```

可视化

> - 方法描述

    - 每一个时刻的task数组对应一个状态。可以自动切换状态，或者通过点击按钮来手动切换状态。
    - task数组里的每一个进程对应屏幕上的一个图形。
        - 这些图形可以是根据在task数组里的位置线性排列，也可以是按照father字段树状排列。
        - 进程不同的状态对应不同的图形，在图形上面会有该进程的pid，如果该进程是当前执行的进程，那么这个pid是红色，否则是白色。
        - 进程的counter越大则图形的颜色更偏暖色调，就更靠近红色。
        - 操作系统执行的不同的操作会对应不同的背景颜色（fork，exit，sched），这样当背景颜色变化的时候就说明执行了不同的操作。
        - 可以点击一个进程来查看进程的详细信息。
- 数据转化
  用Python转output.txt.换成一系列json文件（1.py），以及包涵一些全局信息的configure.json文件。



一部分json数据

```
//configure.json
{
  "state": ["0","3","2","1"],//进程所有可能的状态
    /*
    * 3:fork
    * 4:sched
    * 5:exit
```

```
    */
  "op": ["5","3","4"],//所有可能的操作
  "counter":
["20","18","10","7","4","1","5","25","14","6","21","13","16","15","28","8","2

  ,"0","17","9","22","3","29","12","11","24","27" ],//所有的counter
  "priority": [ "15"],//所有可能的优先级
  "fileCnt": 98,//所有的时刻数，每一个时刻对应一个file
  "maxOp": 5,//可能的操作对应数字最大的一个
  "minOp": 3,//可能的操作对应数字最小的一个
  "maxCounter": 29,//最大的counter
  "minCounter": 0,//最小的counter
  "maxPrioirity": 15,//最大的prioirty
  "minPrioirity": 15//最小的priority
}
```

```
//data_0.json
[//当前的task数组
  {//第一个进程
    "op": "3",//当前操作系统执行的操作
    "pid": "0",
    "father": "-1",
    "priority": "15",
    "counter": "14",
    "state": "0",
    "curPid": "0"//当前正在执行的进程的pid
  },
  {//第二个进程
    "op": "3",
    "pid": "1",
    "father": "0",
    "priority": "15",
    "counter": "15",
    "state": "0",
    "curPid": "0"
  }
]
```

- 用processing可视化：主要的代码结构

```
//Computer.pde
Os os;
void setup(){
  os = new Os();
}

void draw(){
```

```
//操作系统开始执行
  os.run();
}
```

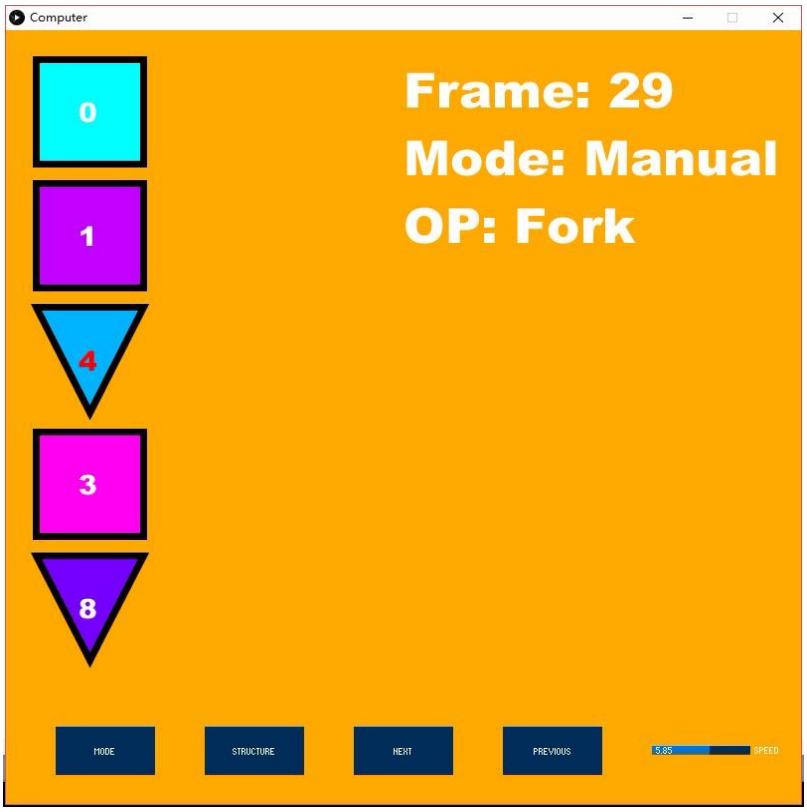
```
//Os.pde
class Os{
    //表示task数组
    ArrayList<Process> proList;

    Os(){
        proList = new ArrayList<Prcocess>();
    }

    void run(){
        update();//更新状态
        display();//渲染屏幕
    }

    void update(){
    }

    void display(){
    }

}
```

```
//Prcocess.pde
class Process{
    Process(){
    }

    void display(){//渲染屏幕
    }

    void update(JSONObject data){//根据data更新进程信息
    }
```

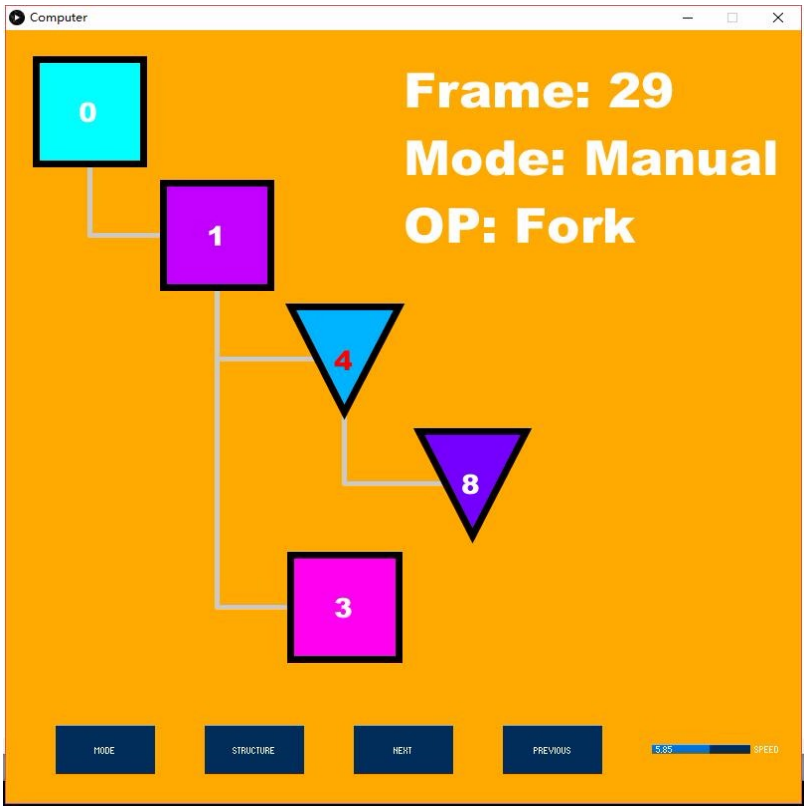
```
//Tree.pde
class Tree{//用于将进程按照树状结构组织
}

class Node{//每一个节点
    Process data;//数据为一个process
}
```

实验结果

线性结构和树状结构

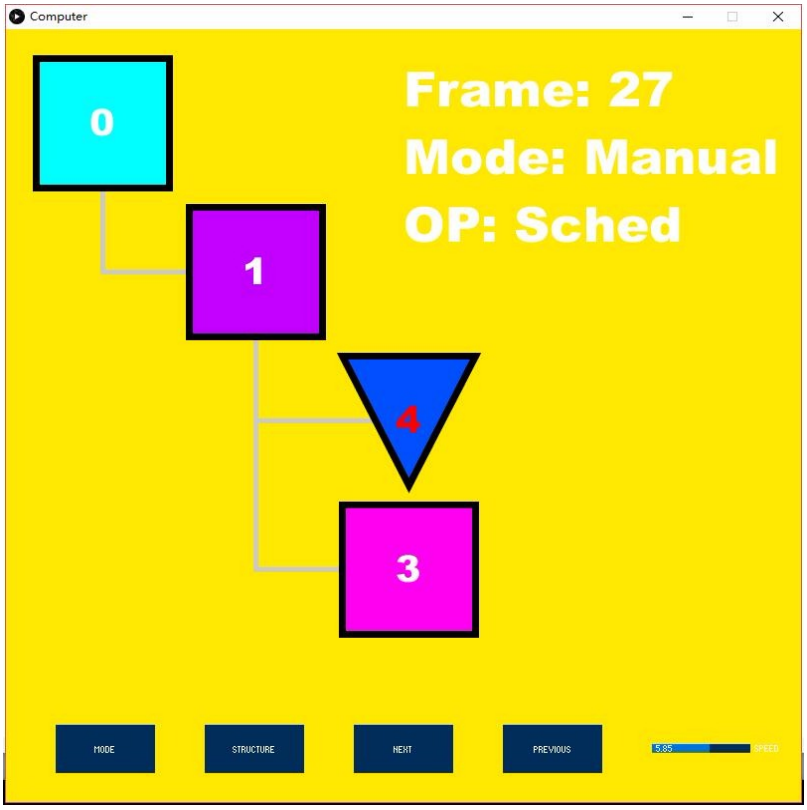


线性结构

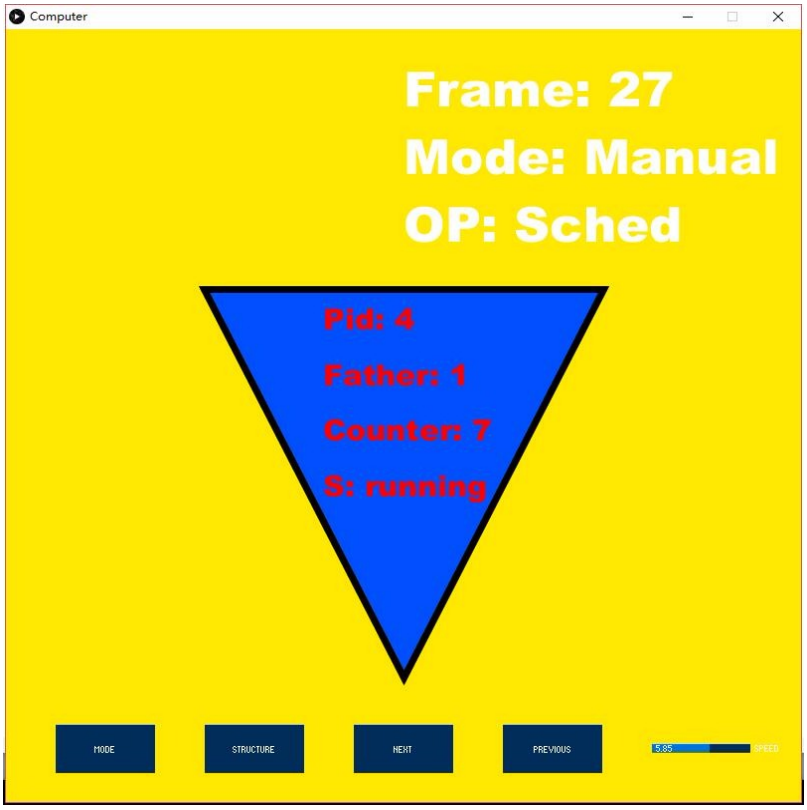


树状结构

全局模式和局部模式



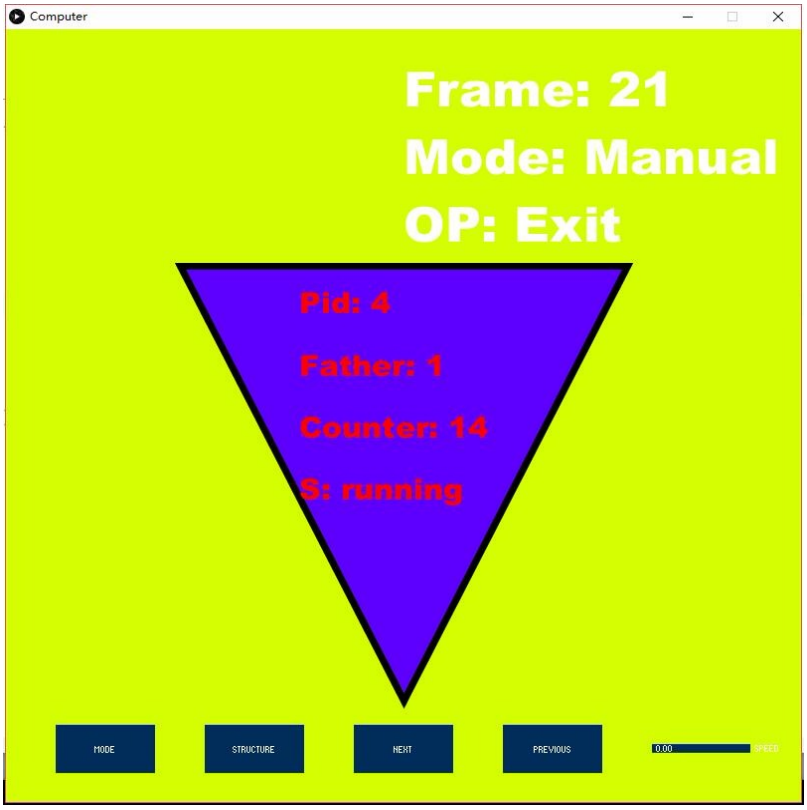
全局模式



局部模式

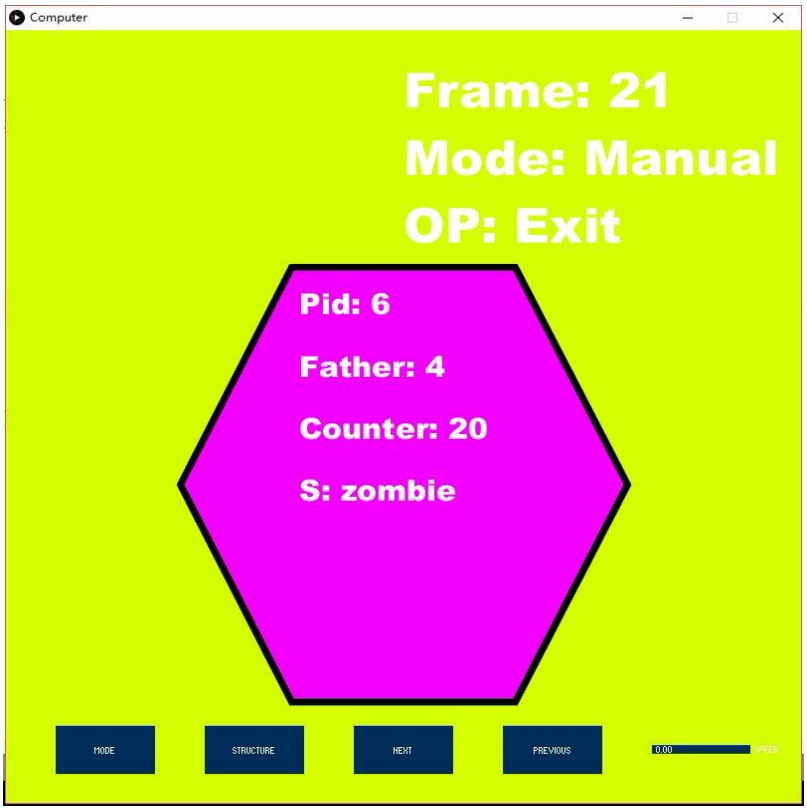
进程不同的状态

> 倒三角表示running状态。



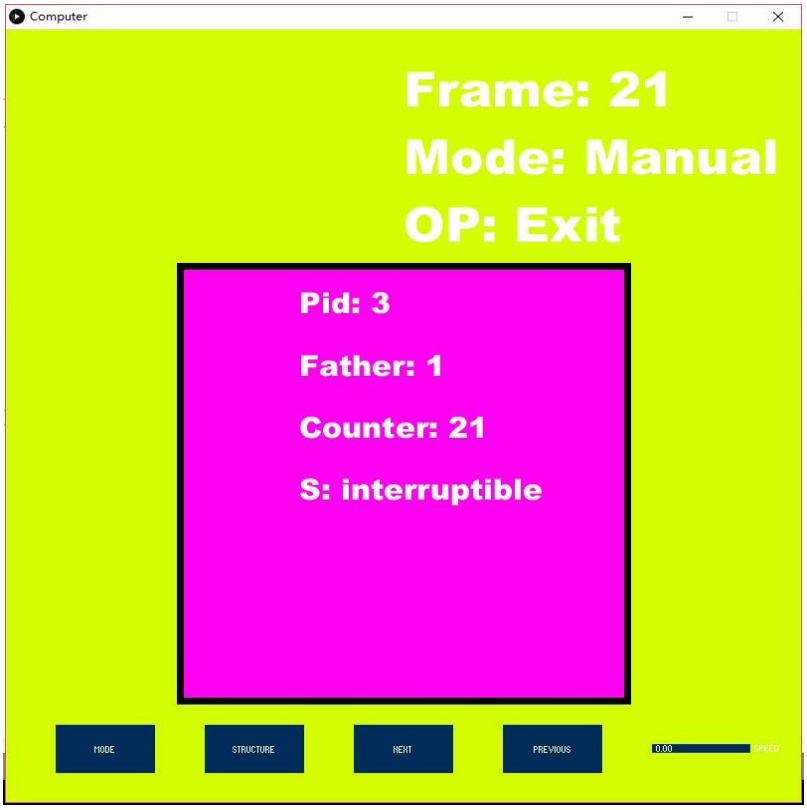
running

> 六边形表示zombie状态。



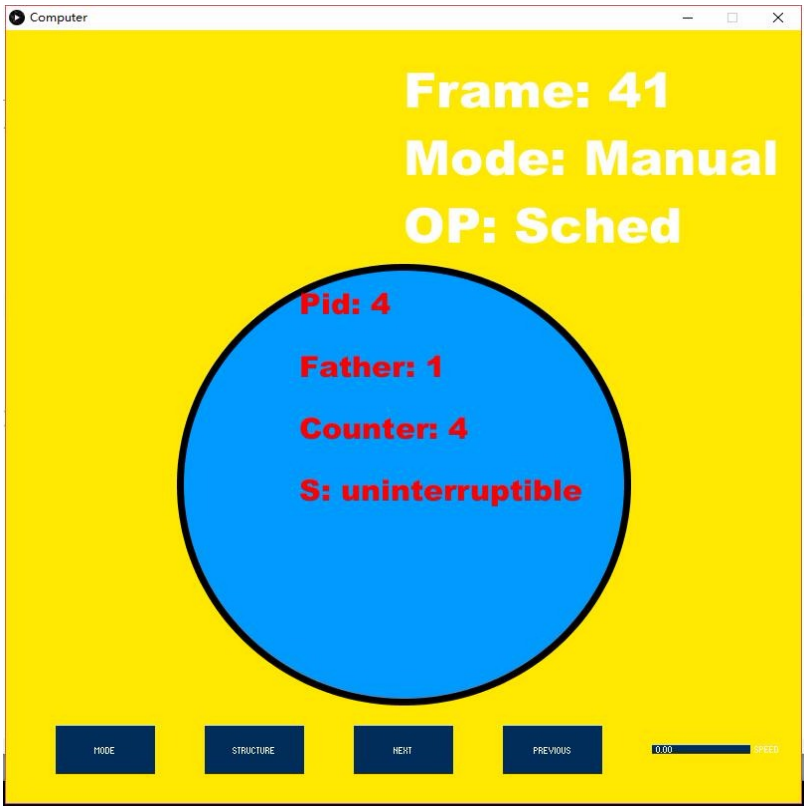
zombie

> 正方形表示interruptible状态。



interruptible

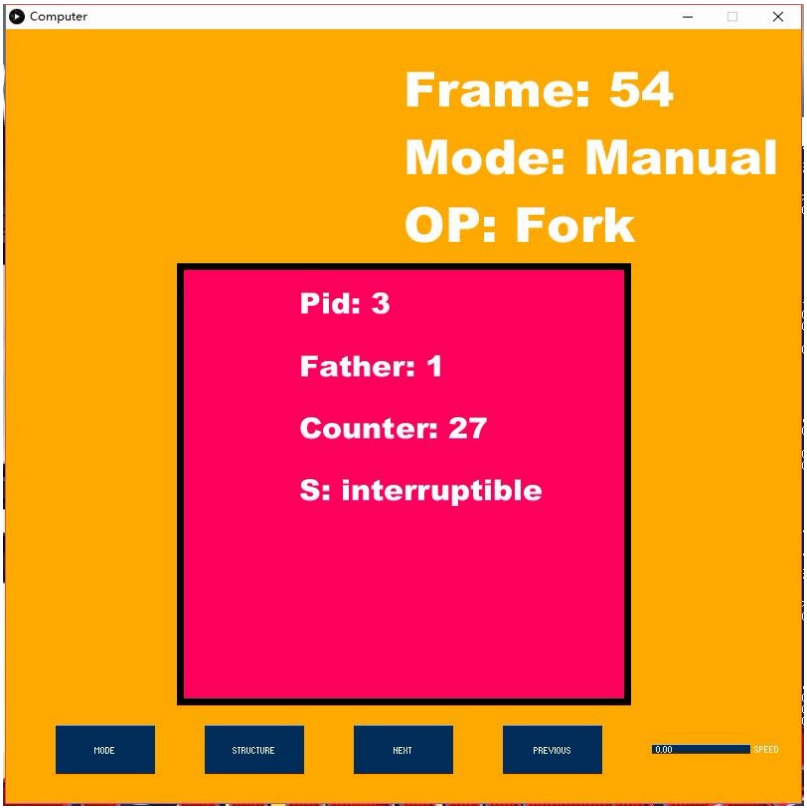
> 圆形表示uninterruptible状态。



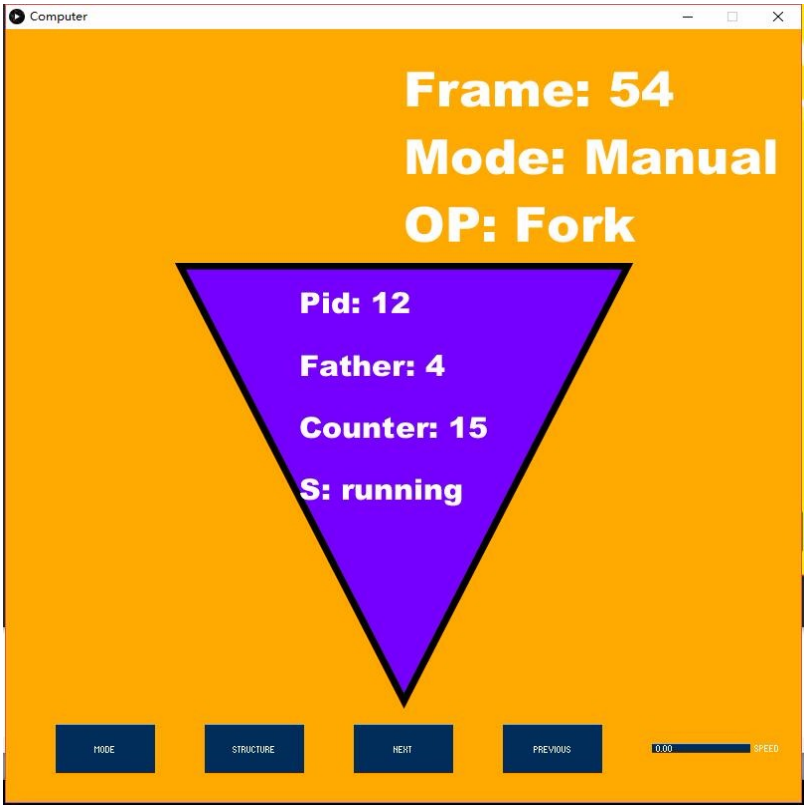
uninterruptible

进程间counter关系

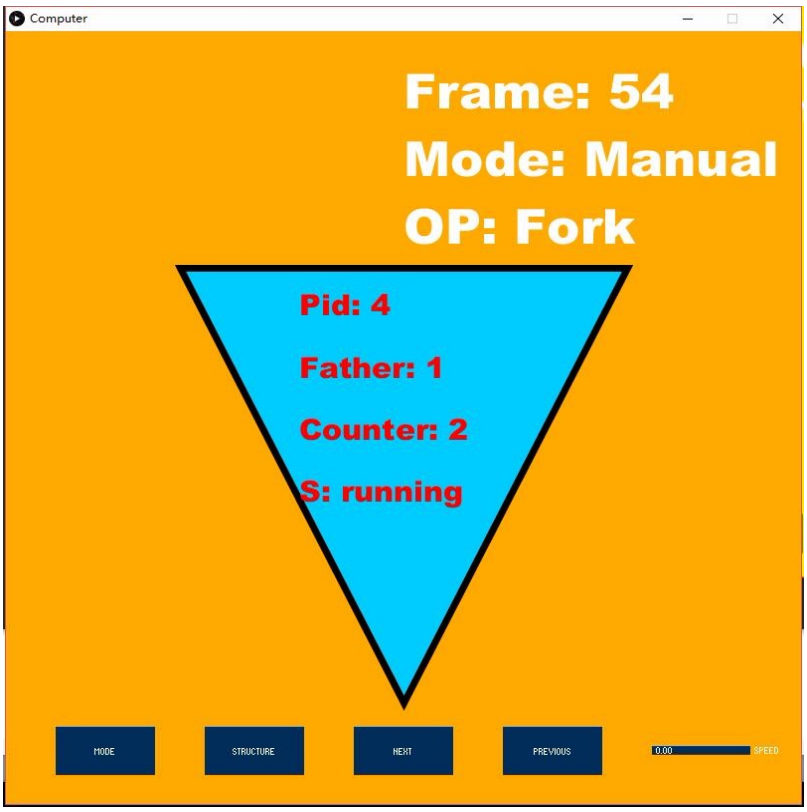
> 下面的图片展示counter和颜色的关系。



counter最大，所以是红色



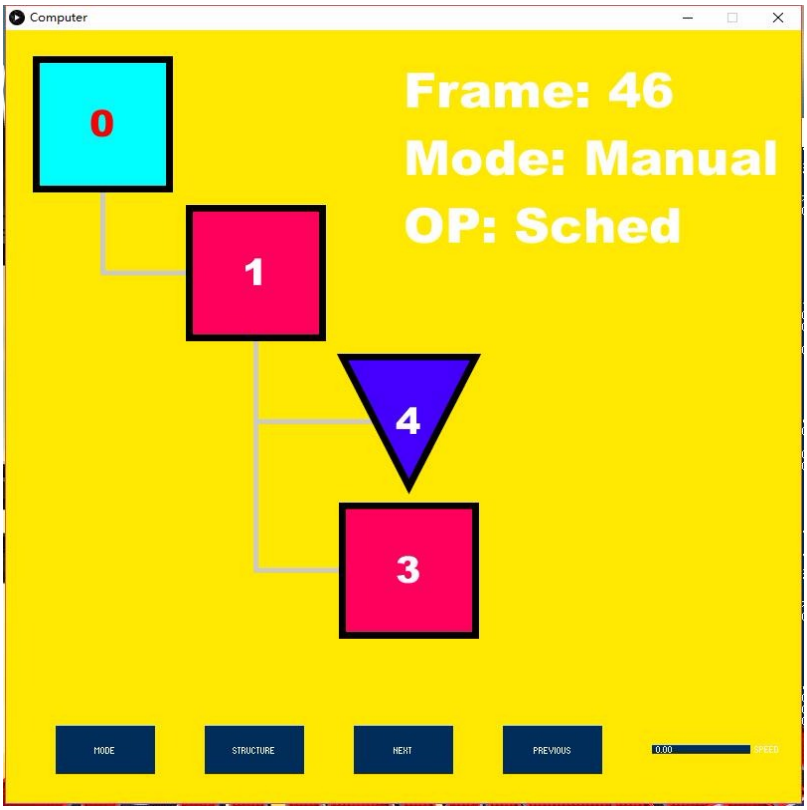
counter次之，所以是紫色



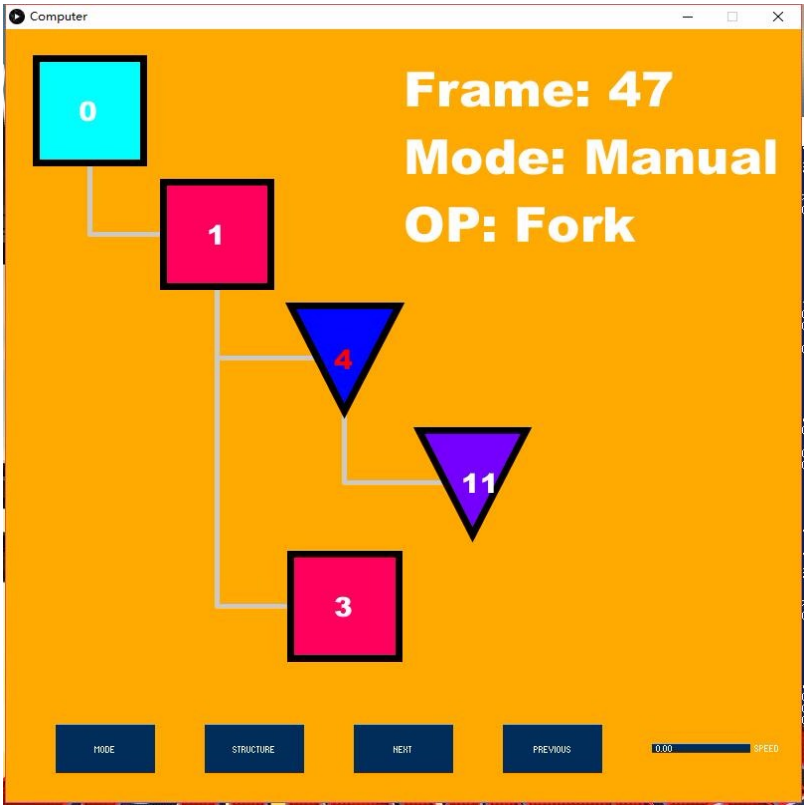
counter最小，所以是浅蓝色

执行不同的操作

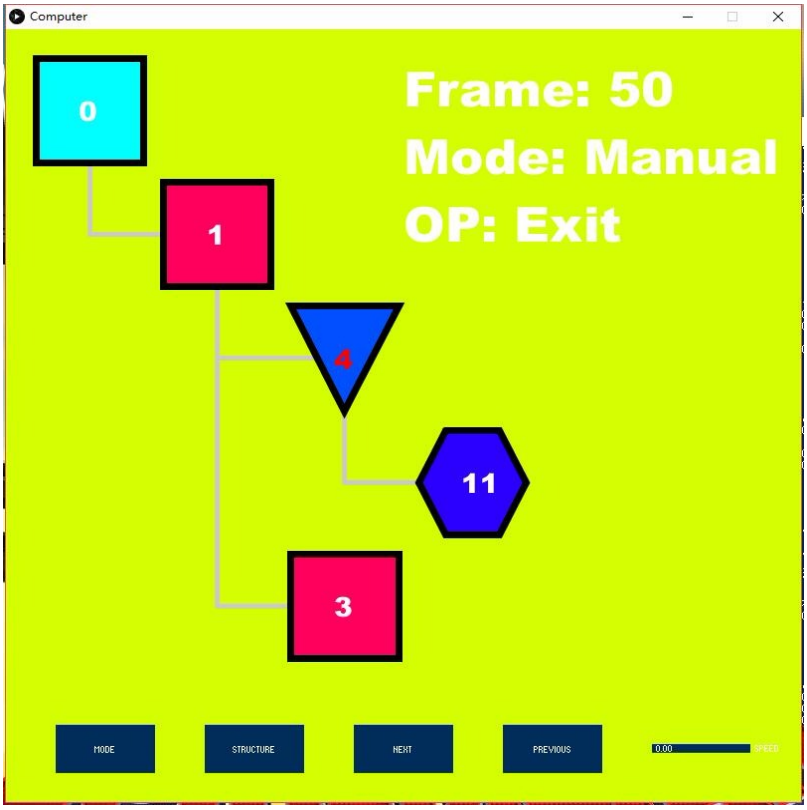
> 下面的图片展示操作系统的fork，sched，exit操作。



从0号进程切换成4号进程，因为是sched操作，所以背景是黄色



4号进程创建出11号进程，因为是fork操作，所以背景是橙色



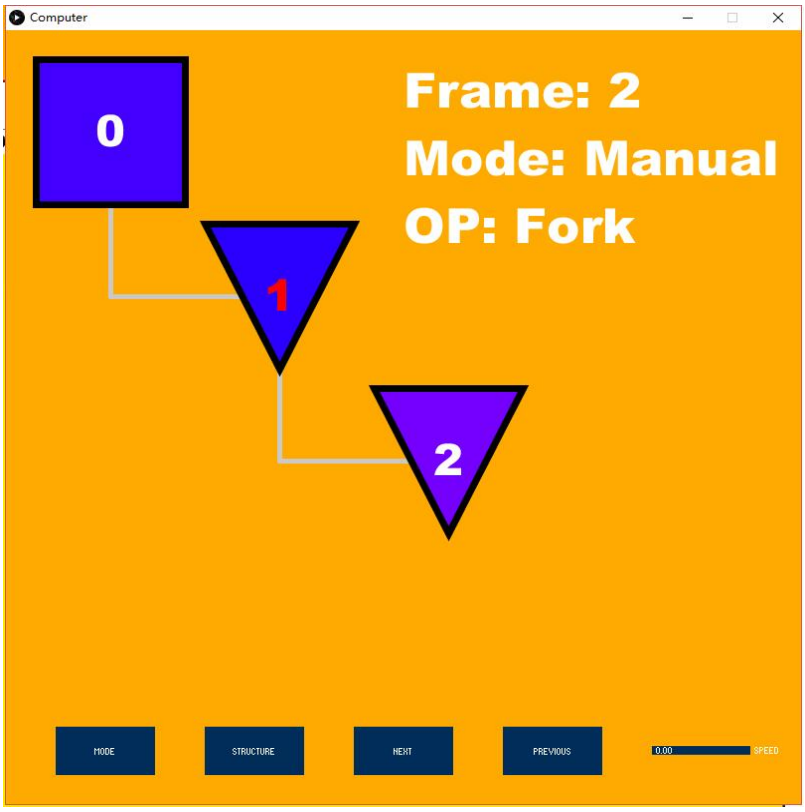
退出11号进程，因为是exit操作，所以背景是嫩绿色

讨论

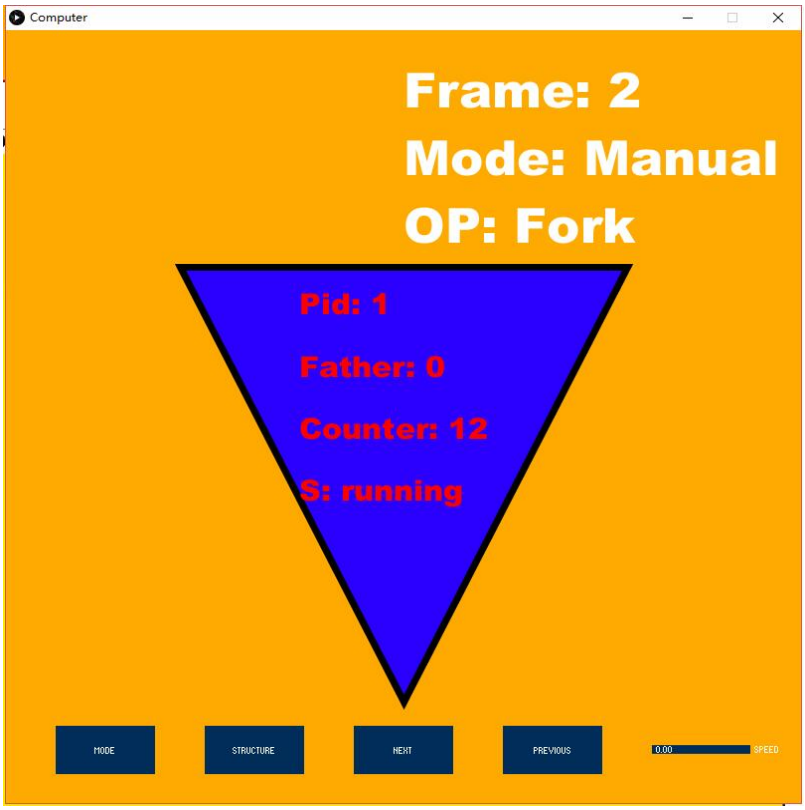
主要可视化了一个进程的生命周期。以及相应的调度算法。

创建一个进程

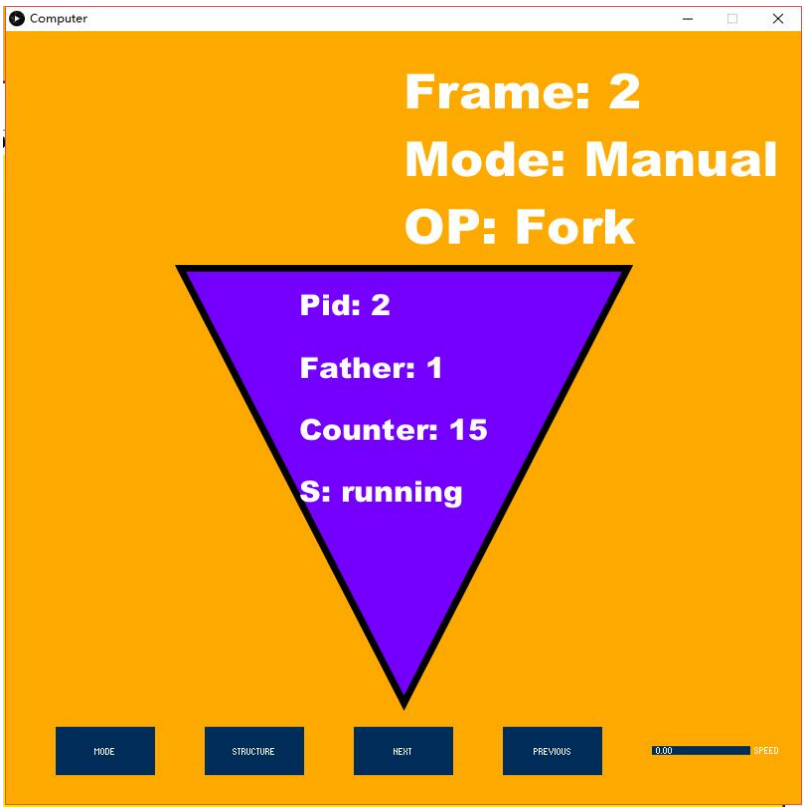
创建一个进程是从父进程复制一个子进程，除了id号、father、counter不一样，其他都一样（状态、priority）。



id不一样。因为父节点不同，所以father也不一样。图形的形状相同，所以状态一样。



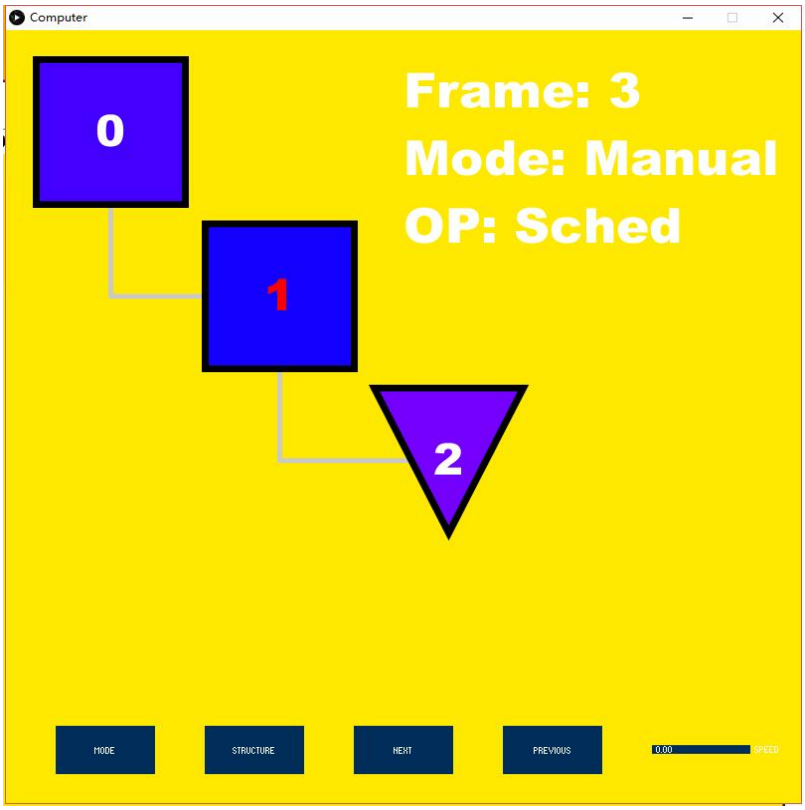
父进程1的counter是12，比创建出来的进程的counter小。



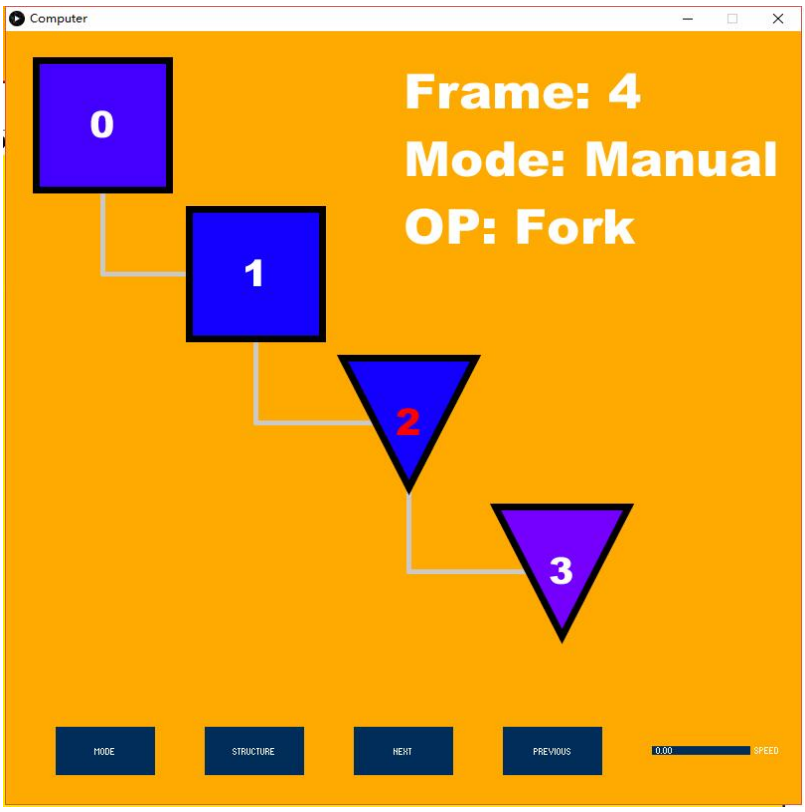
创建出来的进程2的counter是15，比父进程的counter大。

调度一个进程

创建完成后，因为子进程的counter比父进程1的counter大。所以调度到子进程2。此时父进程1从running状态切换成interruputible，然后将子进程2切换成running。



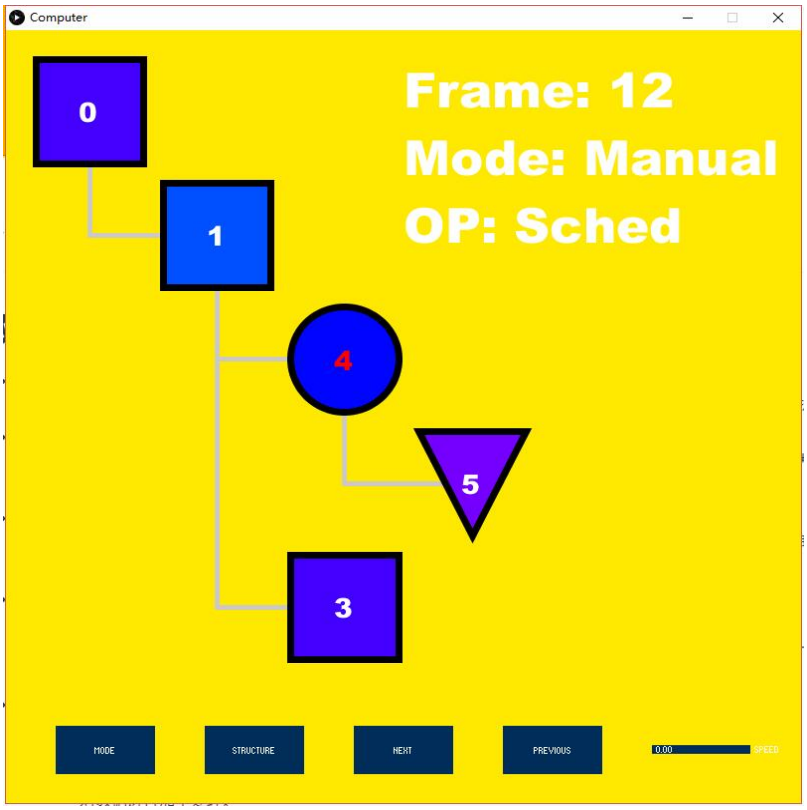
父进程1从三角形变成正方形，所以从running状态切换到interruputible。



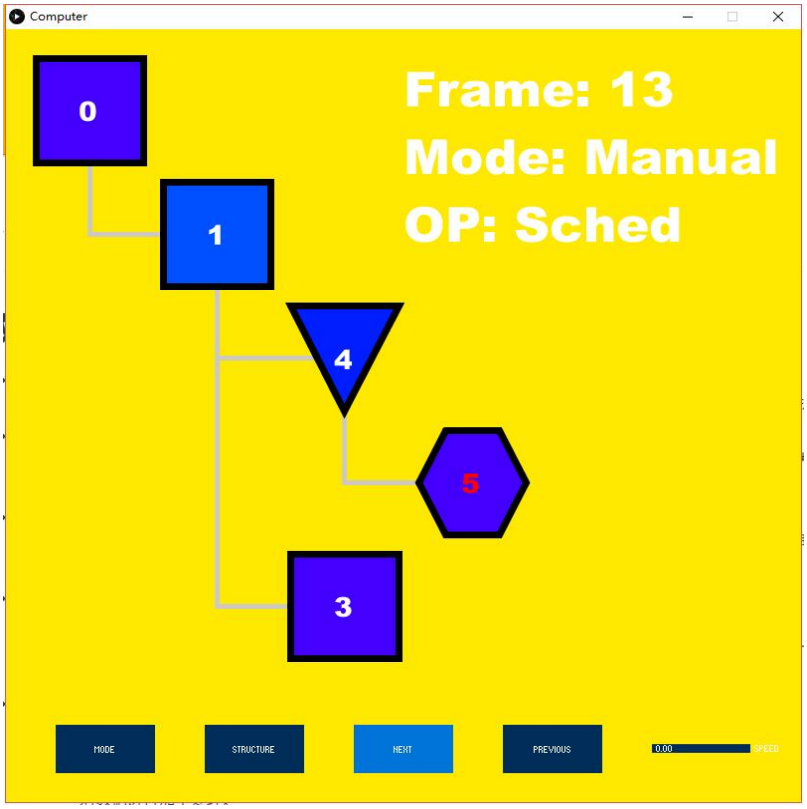
子进程2的id号是红色的，说明为当前执行的进程。

退出一个进程

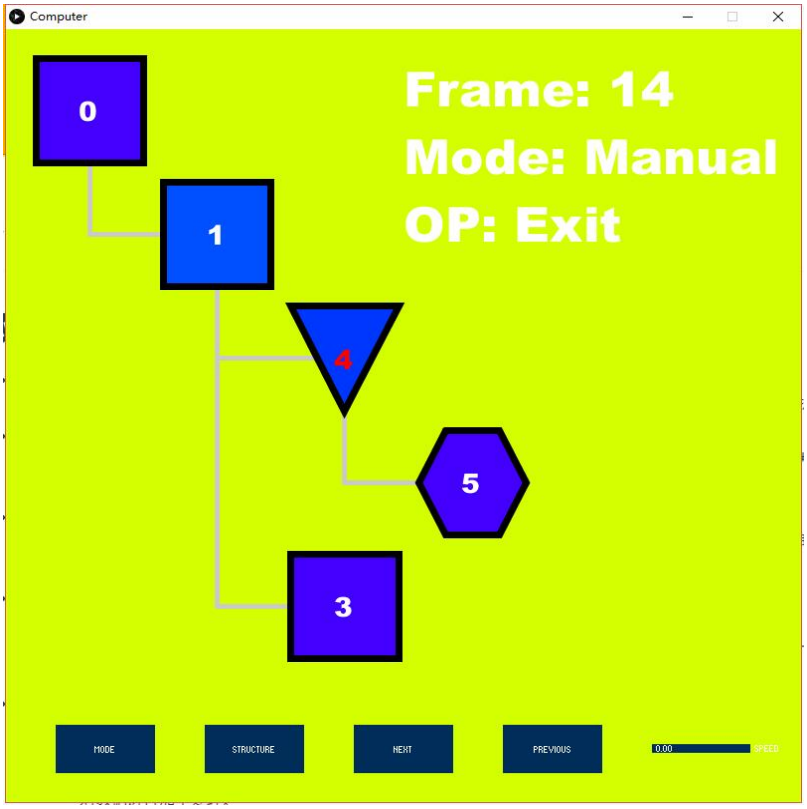
退出一个进程的时候首先执行它的父进程，父进程变成uninterruputible状态。然后切换回该进程，将该进程变成僵尸状态。之后又切换回父进程，最后释放该进程。



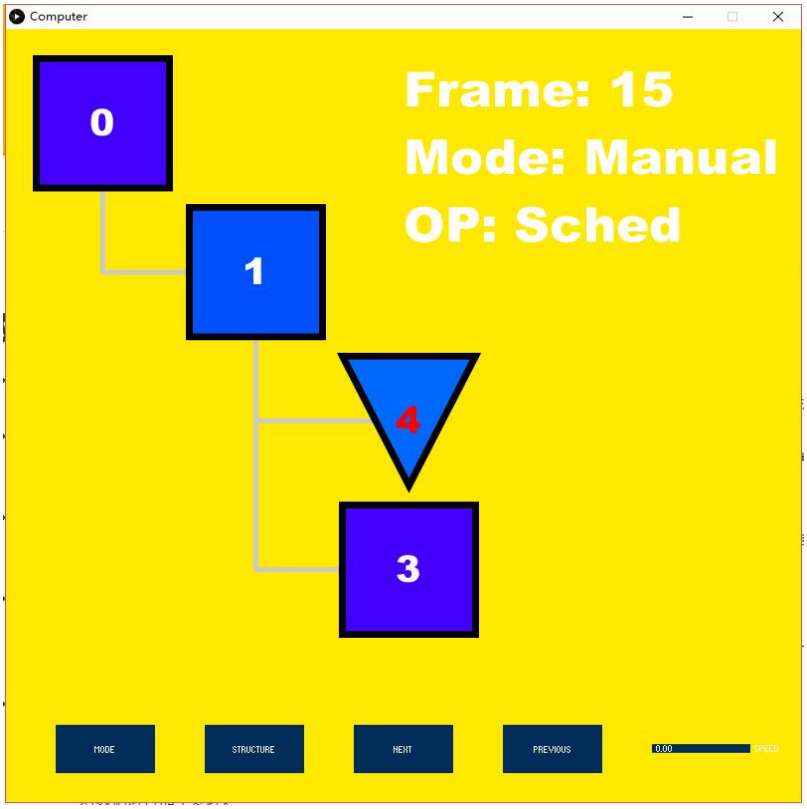
退出一个进程5之前，该进程的父亲4进程变成圆形，说明切换成uninterruputible状态。



进程5的pid变成了红色，说明现在进程5为当前执行的进程。又因为它变成了六边形，说面是一个僵尸进程。



进程4的id变成了红色，说明为当前执行的进程，此时因为是三角形，所以状态是running。

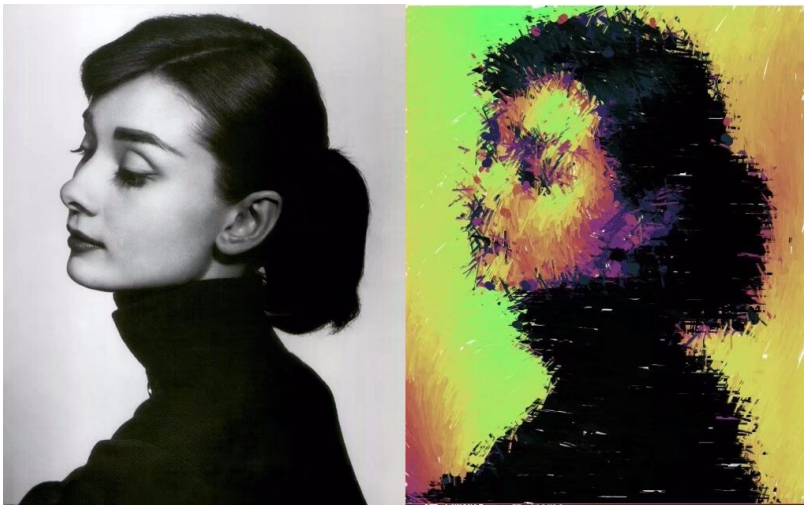


因为上一个状态进程4执行对进程5执行了exit操作，所以进程5消失了。

收获

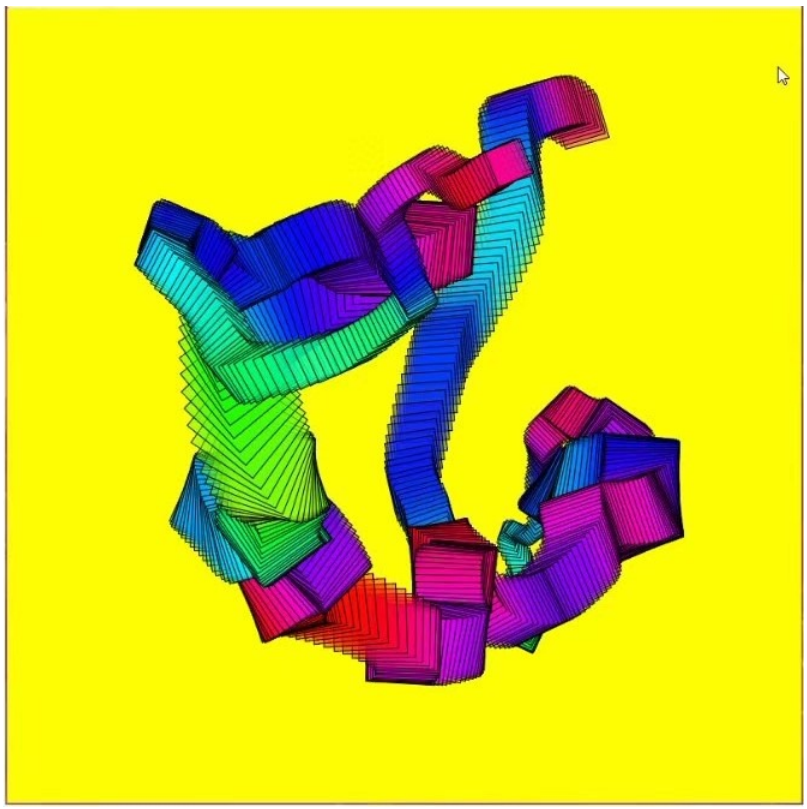
- 更加深入的了解操作系统。
  - 主要是通过分析代码知道操作系统并没有自己原来想的那么复杂，也只是通过一个简简单单的main函数开始执行。
- 提升了沟通和协作写代码，做项目的能力。
  - 和同组者分工很明确。在开始写代码之前，我们约定了数据格式以及相应的接口，然后两边同时开发，这样大大提高了开发的速率。
- 提高了看源代码的能力。
  - 获得的一点心得就是应该自上而下看代码。首先看main函数，大概了解程序是做什么的，然后再看相应的辅助模块，最后挑选一个你最感兴趣的模块着重地看。
- 提升了写代码的能力。
  - 一个函数和变量的命名规范很重要，好的命名比注释更清晰。
  - 写代码的时候也应该自上而下的写。不要一开始就注意实现的细节，应该先把大体的框架搭建起来，然后再一点点的添加。
- 提高了通过可视化数据获得信息的能力。
  - 可视化的正确步骤应该是：
    - 首先确定你想获得信息或者故事。
    - 然后根据你想获得信息去获取相应的数据。
    - 对数据进行加工处理。
    - 开始可视化，并且分析信息。如果获得信息不够或者和预期不相符，那么就重新反复迭代。

- 了解了processing这门语言：
  - 它是基于Java开发的，对于会Java的同学来说，上手很快。
  - processing除了用于可视化数据之外，还有一个很重要的用途：创意编程。所以在学习processing的过程中，我也学习了很多创意编程的东西。 下面是我的一些简单作品。

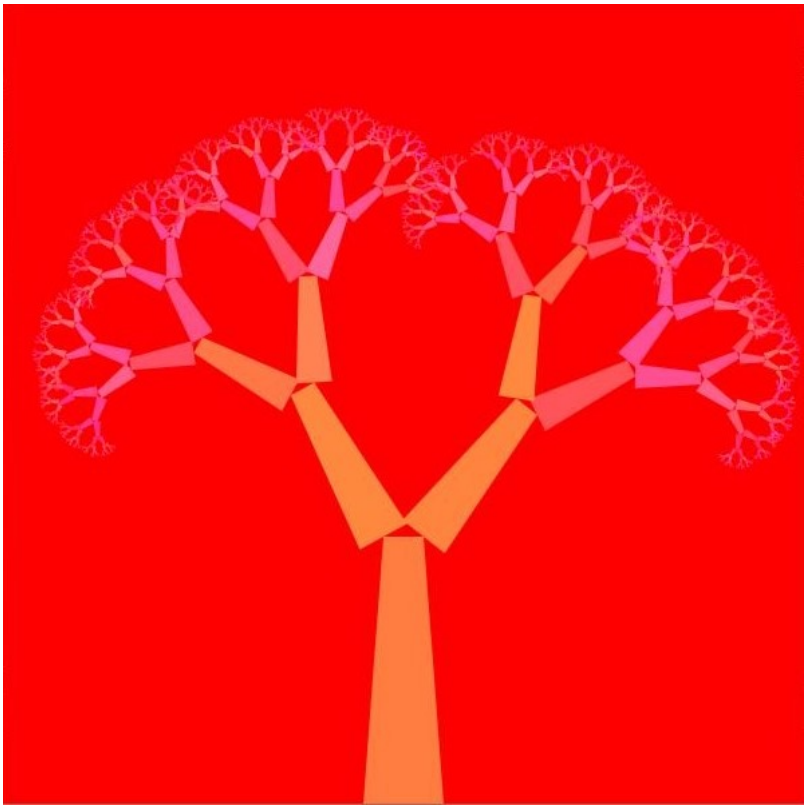


将原图像以一种特别的笔触去渲染

受到《蜘蛛侠：平行宇宙》启发做的天气系统



一条五彩斑斓的龙



火树银花