山东大学
SHANDONG UNIVERSITY

Linux 0.11 可视化

实验报告

姓名：张延慈

同组：张童 陆宇霄

班级：2016 级泰山学堂

学号：201600301320

实验要求

阅读 linux 源码，选择感兴趣部分仔细研究，绘制关键帧，并可视化其过程。

实验目的

正如Linux 系统创始人在一篇新闻组投稿上所说的，要理解一个软件系统的真正运行机制，一定要阅读其源代码(RTFSC – Reading The F**king Source Code)。系统本身是一个完整的整体，具有很多看似不重要的细节存在，但是若忽略这些细节，就会对整个系统的理解带来困难，并且不能真正了解一个实际系统的实现方法和手段。只有在详细阅读过完整的内核源代码之后，才会对系统有一种豁然开朗的感觉，对整个系统的运作过程有深刻的理解。

故在本次实验中，目的是通过阅读Linux 0.11内核的开机启动部分代码和进程部分的代码，来可视化Linux 0.11的进程部分内容，加深对系统运作的理解。

实验环境

硬件环境：计算机一台

软件环境：Linux 0.11 Lab：包括旧的 Linux 内核源代码版本 0.11 和集成的实验环境；processing（用于可视化）

实验过程

实验内容选择

在本次实验中，我们小组选择的内容是进程。主要是因为在系统的内部，对所有事件的处理都是以进程为单位，为进程分配时间片，来完成工作。进程是系统的基础，也是连接内存管理、文件系统的基石。而且进程之间的切换、调度很有趣，在上学期的学习中我们就对调度算法有很大的兴趣，故在这次的课程设计中，我们小组选择了进程；我的主要工作是进程的调度和切换，以及其中的信号量。

实验环境的选择

在本次的实验中，我们有三种环境可以选择：

1、 用 Bochs 仿真系统来跑 Linux 0.11，优点是仿真了 x86 的硬件环境（CPU 指令），缺点是一方面编程体验较差，另一方面操作繁琐需要反复备份，因此很多时间浪费在与操作系统无关的上。

2、 王天浩学长基于 Linux Lab 0.11 开发的实验环境，可以实现对内核的修改，并使用 log 函数输出想要的数据。

3、 陈宇翔同学开发的能够方便使用 gdb 调试的环境

我最终的数据是需要输出进程切换时的进程号、父进程、优先级、切换原因等信息，只有少部分需要查看寄存器，但仍能通过进程的 tss_struct 查看，所有对比所有方法，使用王天浩学长的环境比较方便，故最终选择使用该环境。

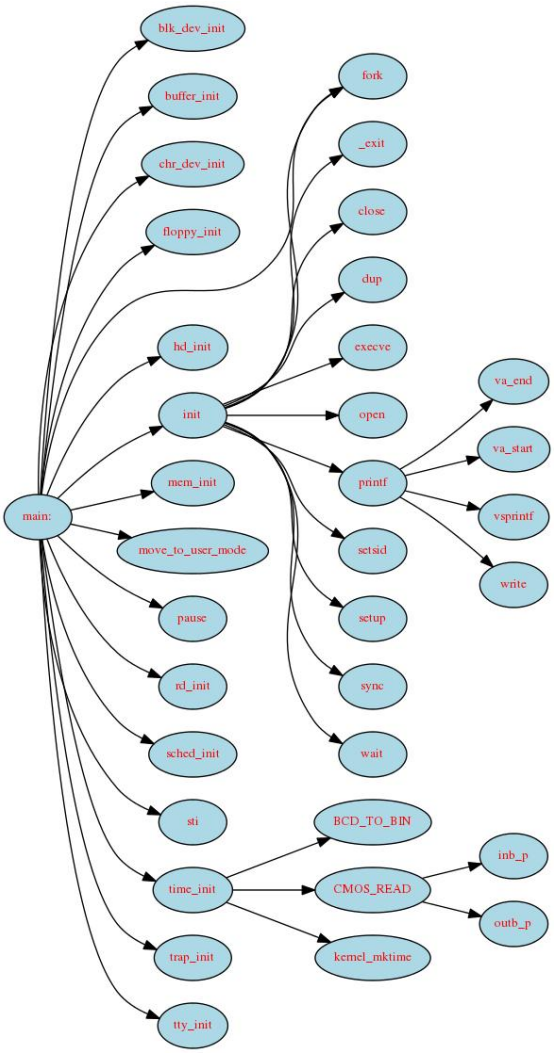
可视化工具的选择

在可视化工具的选择上，由于我们起初的构想是实现一个能够交互的可视化界面，所以开始的打算是使用 PyQt，因为它的信号槽机制能够方便地进行交互。但是后来发现可视化工具与系统输出的数据实时交互有些困难，比如如果希望点击创建进程就能够创建进程的话，需要把可视化程序与 Linux 0.11 的命令行建立连接。所以最后我们还是使用了一种不能实现通过可视化界面创建进程等交互行为的方法，processing。Processing 的特点是直观，易上手，开源，功能强大，通过 processing，我们可以轻松把进程的相关内容做成视频，清晰直观。

实验步骤

阅读源码

首先通读了 Linux 0.11 的源代码，并绘制了整个系统代码的结构图：



然后选择了开机部分的setup.S, head.s, main.c与进程部分的sched.c, signal.c, exit.c 与 fork.c 进行详细阅读。Sched.c 中的内容主要包含进程调度的算法、进程在什么情况下会被切换；signal.c 中主要包含触发 sched.c 中信号是在什么情况下对谁发出的；exit.c 是讲进程的退出；fork.c 是讲进程的创建。

提取数据

我的工作是提取进程的调度部分的数据，故在 sched.c 以及部分 exit.c 的数据中加入 log 函数，来输出想要的数据。

在 sched.c 中，首先修改了 show_task 函数，在其中加入了 log 函数，使能够输出我想要的进程的信息：

```
void show_task(int nr,struct task_struct * p)
{
    //int i,j = 4096-sizeof(struct task_struct);
    //printk("%d: pid=%d, state=%d, ",nr,p->pid,p->state);
    log("%d: pid=%d, state=%d, counter = %d, father=%d, ss = %x, esp = %x, cs = %x, eip = %x \njiffies is %d\n\n",nr,p->pid,p->state, p->counter, p->father, p->tss.ss, p->tss.esp, p->tss.cs, p->tss.eip, jiffies);
    log("\n");
    //i=0;
    //while (i<j && !((char *)(p+1))[i])
    //    i++;
    //printk("%d (of %d) chars free in kernel stack\n\r",i,j);
    //log("%d (of %d) chars free in kernel stack\n",i,j);
}
```

修改后，每一次调用 show_task 函数，都能够显示当前进程的 pid，状态，优先级，父进程，以及调用这个进程时向寄存器压入的信息，如 ss, esp, cs, eip。在 schedule 函数，及使用调度算法的调度函数中，修改信息如下：

```
/* this is the scheduler proper: */
        while (1) {
                c = -1;
                next = 0;
                i = NR_TASKS;
                p = &task[NR_TASKS];
                while (--i) {
                        if (!*--p)
                                continue;
                        if ((*p)->state == TASK_RUNNING && (*p)->counter > c){
                                c = (*p)->counter, next = i;
                                log("Maybe %d is the longest counter and %d is the chosen one?\n", c, task[next]->pid);
                        }
                }
                if (c) break;
                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;
        }
        log("The chosen task:\n");
        show_task(next, task[next]);
        log("all tasks as follows:\n");
        show_stat();
        switch_to(next);
}
```

首先，调度算法是从后往前遍历所有的状态为 1，即运行或运行的进程，找到优先级 counter 最大的进程，选择并切换。于是在遍历过程中，通过 log 输出可能被选中的进程，体现 调度算法在寻找优先级最大的进程的过程。

然后，在 switch_to（next）函数，即切换函数之前，输出被选中的进程的信息以及当前所有进程的信息。

修改完 schedule 函数后，其他函数都是使用 schedule 函数进行调度，所以只需要在其他函数调用 schedule 函数之前，使用 log 函数输出为什么在这里调用 schedule 函数即可。

Sys_pause 函数是系统调用，转换当前任务的状态为可终端的等待状态，并重新调度。该系统调用将导致进程进入睡眠状态，直到收到一个信号，该信号用于终止进程或者使进程调用一个信号捕获函数。只有当捕获了一个信号，并且信号捕获处理函数返回，pause 才会返回。此时 pause 的返回值应该是-1，并且 error 被置为 EINTR，这里还没有完全实现。

```
int sys_pause(void)
{
        current->state = TASK_INTERRUPTIBLE;
        log("Schedule is called by sys_pause.\n");
        schedule();
        log("\n");
        return 0;
}
```

Sleep_on 函数是把状态置为不可中断睡眠状态，并调用调度函数。

```
void sleep_on(struct task_struct **p)
{
        struct task_struct *tmp;

        if (!p)
                return;
        if (current == &(init_task.task))
                panic("task[0] trying to sleep");
        tmp = *p;
        *p = current;
        current->state = TASK_UNINTERRUPTIBLE;
        log("Schedule is called by sleep_on.\n");
        schedule();
        log("\n");
        *p = tmp;
        if (tmp)
                tmp->state=TASK_RUNNING;
}
```

Interruptible_sleep_on 函数是把当前进程的状态置为可中断睡眠状态，并重新调度。

```
void interruptible_sleep_on(struct task_struct **p)
{
        struct task_struct *tmp;

        if (!p)
                return;
        if (current == &(init_task.task))
                panic("task[0] trying to sleep");
        tmp=*p;
        *p=current;
repeat: current->state = TASK_INTERRUPTIBLE;
        log("Schedule is called by interruptible_sleep_on.\n");
        schedule();
        log("\n");
        if (*p && *p != current) {
                (*p)->state = TASK_RUNNING;
                goto repeat;
        }
        *p = tmp;
        if (tmp)
                tmp->state = TASK_RUNNING;
}
```

Wake_up 函数是唤醒正处于睡眠状态的进程。

```
void wake_up(struct task_struct **p)
{
        if (p && *p) {
                (*p)->state = TASK_RUNNING;
                *p = NULL;
        }
}
```

Do_timer 函数是当分给每一个进程的时间片耗尽时使用的函数，该函数的作用就是重现调度，并重置时间片计时器。

```
void do_timer(long cpl)
{
        extern int beepcount;
        extern void sysbeepstop(void);

        if (beepcount)
                if (!--beepcount)
                        sysbeepstop();

        if (cpl)
                current->utime++;
        else
                current->stime++;

        if (next_timer) {
                next_timer->jiffies--;
                while (next_timer && next_timer->jiffies <= 0) {
                        void (*fn)(void);

                        fn = next_timer->fn;
                        next_timer->fn = NULL;
                        next_timer = next_timer->next;
                        (fn)();
                }
        }
        if (current_DOR & 0xf0)
                do_floppy_timer();
        if ((--current->counter)>0) return;
        current->counter=0;
        if (!cpl) return;
        log("schedule is called by do_time because the user time slice is used out.\n");
        schedule();
        log("\n");
}
```

以上是对 sched.c 函数的所有修改。

在 exit.c 函数中，同样使用了进程调度函数 schedule。

Release 函数是释放指定进程占用的任务槽及其任务数据结构占用的内存页面。释放后使用 schedule 函数进行重新调度。

```
void release(struct task_struct * p)
{
        log("realse start:\n");
        int i;

        if (!p)
                return;
        for (i=1 ; i<NR_TASKS ; i++)
                if (task[i]==p) {
                        log("tast[%d] is destroyed\n",i);
                        task[i]=NULL;
                        log("tast[i] is availible now: ");
                        for(i=1 ; i<NR_TASKS ; i++)
                                if (!task[i]){
                                        log("%d ",i);
                                }
                        log("\n");
                        free_page((long)p);
                        log("Schedule is called by release.\n");
                        schedule();
                        log("\n");
                        return;
                }
        panic("trying to release non-existent task");
}
```

Do_exit 函数是程序退出处理函数，会被 sys_exit 调用，该程序将根据当前进程自身的特性对其进行处理，并把当前进程状态设置为 3 状态，即僵尸状态，最后调用调度函数 schedule 去执行其他进程，不再返回。

```
int do_exit(long code)
{
        log("do_exit start:\n");
        int i;
        free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
        free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
        for (i=0 ; i<NR_TASKS ; i++)
                if (task[i] && task[i]->father == current->pid) {
                        log("change task[%d]'s father from %d to pid_1\n",task[i]->father,i);
                        task[i]->father = 1;
                        if (task[i]->state == TASK_ZOMBIE)
                                /* assumption task[1] is always init */
                                (void) send_sig(SIGCHLD, task[1], 1);
                }
        for (i=0 ; i<NR_OPEN ; i++)
                if (current->filp[i])
                        sys_close(i);
        iput(current->pwd);
        current->pwd=NULL;
        iput(current->root);
        current->root=NULL;
        iput(current->executable);
        current->executable=NULL;
        if (current->leader && current->tty >= 0)
                tty_table[current->tty].pgrp = 0;
        if (last_task_used_math == current)
                last_task_used_math = NULL;
        if (current->leader)
                kill_session();
        current->state = TASK_ZOMBIE;
        current->exit_code = code;
        tell_father(current->father);
        log("do_exit call schedule~\n");
        log("Schedule is called by do_exit.\n");
        schedule();
        log("\n");
        return (-1);    /* just to suppress warnings */
}
```

最后一部分使用 schedule 调度函数的是 sys_waitpid，系统调用 wait_pid，挂起当前进程，直到 pid 指定的子进程退出（终止）或者收到要求终止该进程的信号，或者需要调用一个信号句柄（信号处理程序）。如果 pid 所指的子进程早已退出（已成为僵尸进程），则本调用将立刻返回，子进程使用的所有资源将释放。

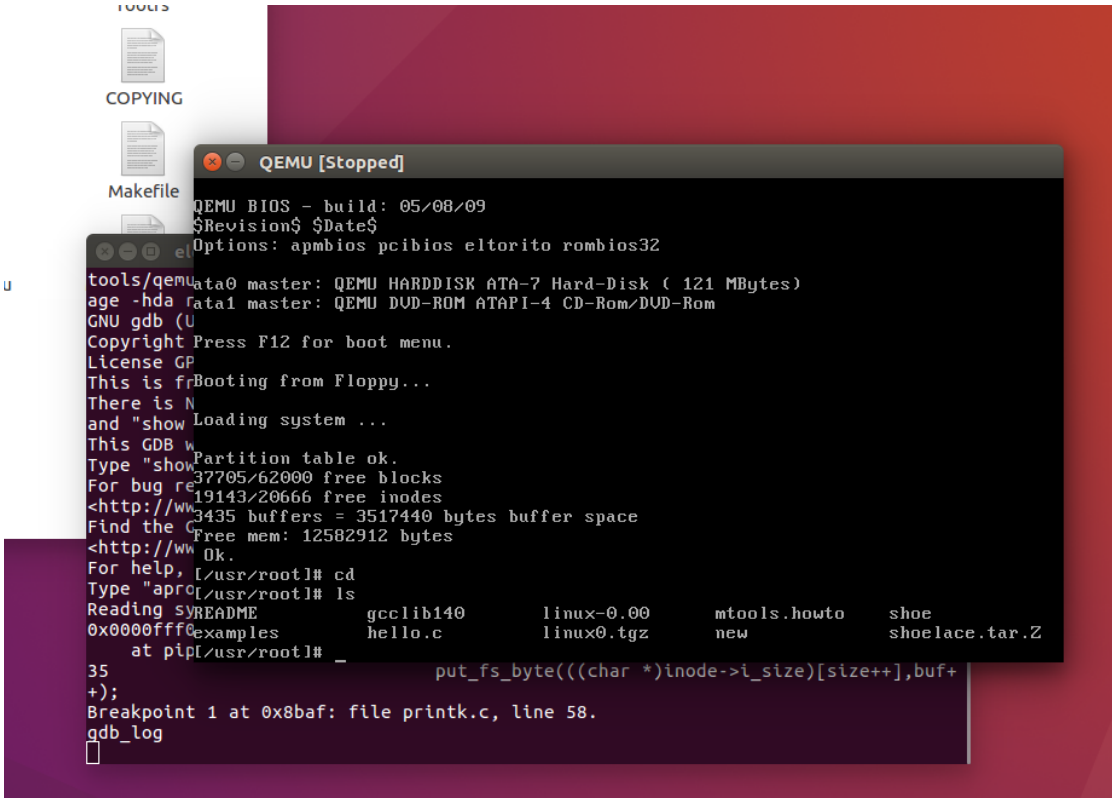
```
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
        log("sys_waitpid start:\n");
        int flag, code;
        struct task_struct ** p;

        verify_area(stat_addr,4);
repeat:
        flag=0;
        for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
                if (!*p || *p == current)
                        continue;
                if ((*p)->father != current->pid)
                        continue;
                if (pid>0) {
                        if ((*p)->pid != pid)
                                continue;
                } else if (!pid) {
                        if ((*p)->pgrp != current->pgrp)
                                continue;
                } else if (pid != -1) {
                        if ((*p)->pgrp != -pid)
                                continue;
                }
                switch ((*p)->state) {
                log("waitpid_state:%d\n",(*p)->state);
                        case TASK_STOPPED:
                                if (!(options & WUNTRACED))
                                        continue;
                                put_fs_long(0x7f,stat_addr);
                                return (*p)->pid;
                        case TASK_ZOMBIE:
                                current->cutime += (*p)->utime;
                                current->cstime += (*p)->stime;
                                flag = (*p)->pid;
                                code = (*p)->exit_code;
                                release(*p);
                                put_fs_long(code,stat_addr);
                                return flag;
                        default:
                                flag=1;
                                continue;
                }
        }
```

```
        if (flag) {
                if (options & WNOHANG)
                        return 0;
                current->state=TASK_INTERRUPTIBLE;
                log("set unable to be break\n");
                log("Schedule is called by sys_waitpid.\n");
                schedule();
                log("\n");
                log("waitpid call schedule~\n");
                if (!(current->signal &= ~(1<<(SIGCHLD-1))))
                        goto repeat;
                else
                        return -EINTR;
        }
        return -ECHILD;
}
```

以上是为了提取数据对所有代码部分的修改。

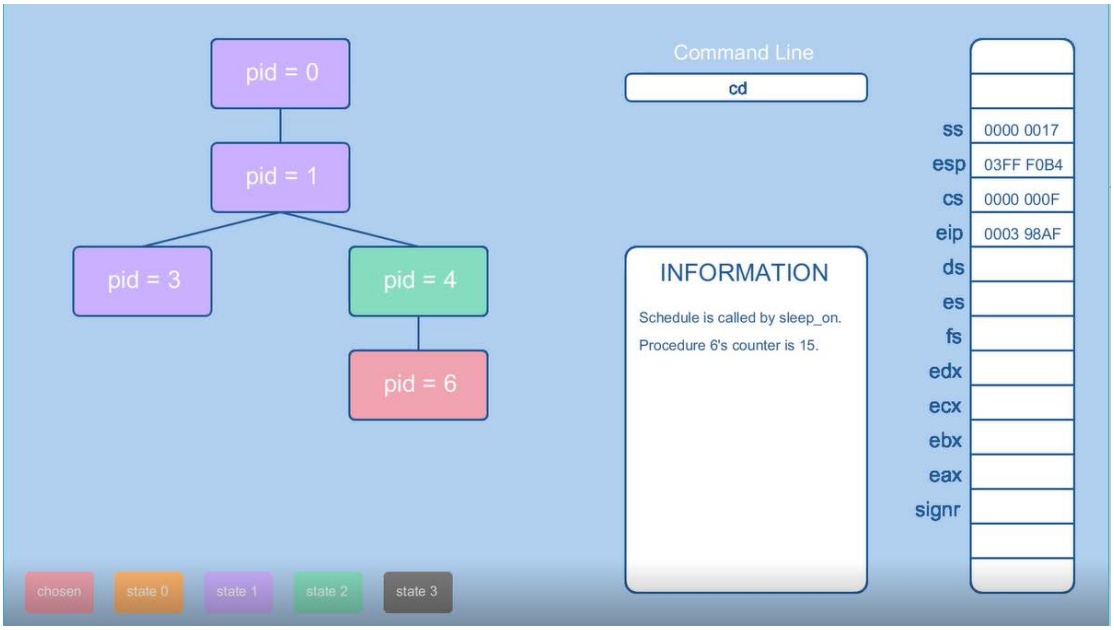
为了提取进程切换的数据，除了系统开机时启动的 5 个进程，还在 qemu 中使用指令创建进程。实验中使用的指令是 cd 和 ls，其中 cd 指令会创建一个进程，ls 指令会创建两个进程。



可视化数据

实验使用 processing 进行可视化，具体对 log 函数的输出结果 output.txt 进行可视化，ouput.txt 中的信息包括进程创建和销毁时在进程树上的信息和进程调度时在进程树上的信息以及调度的原因、在寄存器值上的体现。

可视化时，首先构造了可视化界面的基本框架：



可视化的主要界面包括五部分：

左边是按照进程切换的顺序绘制的进程树，进程树表明进程的父子关系以及各个进程的 pid

左下部是进程五个状态的图例

中间上部是使执行图示的进程切换的进程的指令

中间下部是进程切换的信息，以及被选中进程的优先级

右部是进程切换对寄存器进行的压栈、保存原有状态的信息。

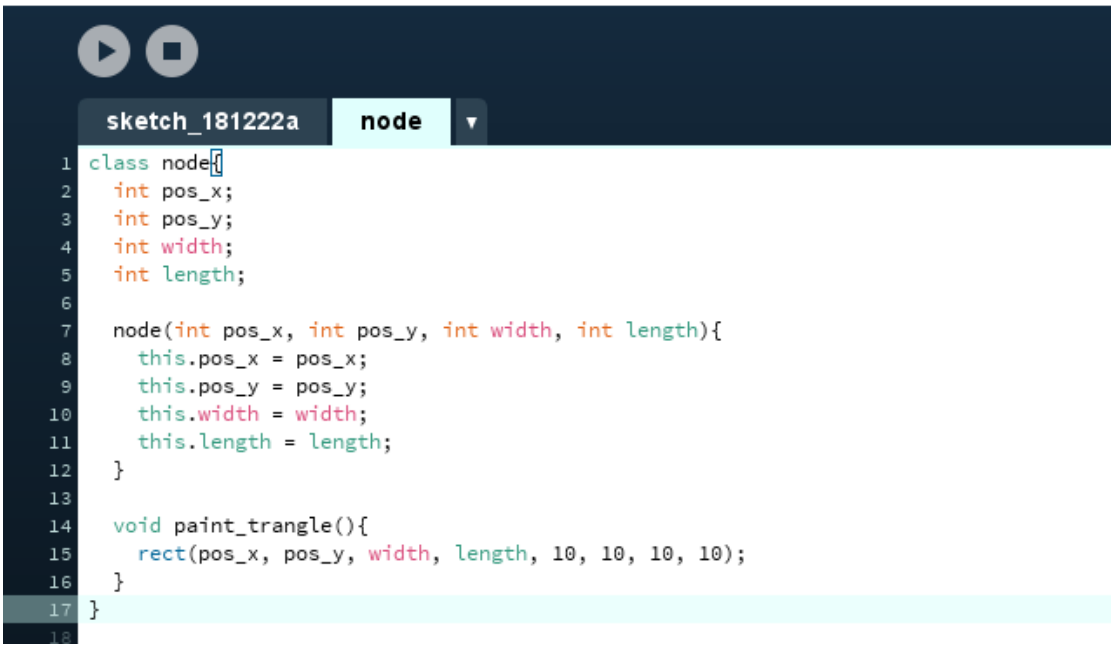
图例为：

linux 系统中，一个进程有 5 种可能状态，在 sched.c 第 19 行处定义了状态的标识：

```
#define TASK_RUNNING      0    // 正在运行或可被运行状态
#define TASK_INTERRUPTIBLE    1    // 可被中断睡眠状态
#define TASK_UNINTERRUPTIBLE    2    // 不可中断睡眠状态
#define TASK_ZOMBIE   3    // 僵死状态
#define TASK_STOPPED 4    // 停止状态
```

其中，4 状态在 linux 0.11 中没有使用

为了搭建这个基本框架以便以后对进程树、信息和寄存器等信息进行操作，给节点专门用了一个类，来方便节点的绘制：



然后初始化背景颜色、每一个区域的位置等基本信息（以下为代码部分节选，完整代码见附件）

```
color color5 = color(153,204,255);
color color6 = color(255,204,238);
color color7 = color(68,187,187);
color color8 = color(217,191,244);
color chosen = color(233, 155, 177);
color stat0 = color(255, 177, 112);
color stat1 = color(202, 170, 251);
color stat2 = color(142, 232, 193);
color stat3 = color(126, 126, 126);
color green = stat2;
int main_memory_x = 625;
int main_memory_y = 100;
int main_memory_w = 250;
int main_memory_l = 700;
void setup(){
  smooth(4);
  background(color1);
  size(1600, 900);
//  frameRate(4);
  for(int i=0;i<36;i++){
    event_time[i] = 2000*(i+1);
  }
  event_time[36] = 2000*37+23000;
  for(int i=37;i<100;i++){
    event_time[i]=23000+2000*(i+1);
  }
}
```

按照创建、切换、销毁进程的顺序，写每一次事件发生时的代码：

```
  if(238000 <= time && time <= 241000){
    start = 238000;
    create_proc2(start);
  }
  if(241000 <= time && time <= 244000){
    start = 241000;
    choose_proc2(start);
  }
  if(244000 <= time && time <= 247000){
    start = 244000;
    create_proc3(start);
  }
  if(247000 <= time && time <= 250000){
    start = 247000;
    exit_proc2(start);
  }
  if(250000 <= time && time <= 286000){
    start = 250000;
    delete_proc(250000);//36s
  }
  if(286000 <= time && time <= 289000){
    start = 286000;
    do_exit_2(start);//change 3's father
  }
  if(289000 <= time && time <= 292000){
    start = 289000;
    line222(start);
  }
```

每一个事件发生时，需要重新绘制全部画布，即用背景全部覆盖原有部分，然后加入这次事件对画布有影响的变化。例如，由释放 5 进程内存空间导致的进程切换：

```
967  void release_5(int start){
968    background(color1);
969    //STACK
970    fill(255);
971    strokeWeight(3);
972    stroke(color3);
973    rect(1400, 50, 150, 800, 20, 20, 20, 20);
974    for(int i=1; i<16; i++){
975      line(1400, 50+50*i, 1550, 50+50*i);
976    }
977    fill(color3);
978    textFont(createFont("Arial", 30, true));
979    text("ss", 1360, 190, -1);
980    text("esp", 1345, 240, -1);
981    text("cs", 1360, 290, -1);
982    text("eip", 1350, 340, -1);
983    text("ds", 1360, 390, -1);
984    text("es", 1360, 440, -1);
985    text("fs", 1365, 490, -1);
986    text("edx", 1340, 540, -1);
987    text("ecx", 1340, 590, -1);
988    text("ebx", 1340, 640, -1);
989    text("eax", 1340, 690, -1);
990    text("signr", 1320, 740, -1);
991    textFont(createFont("Arial", 23, true));
992    text("0000 0010", 1420, 190, -1);
993    text("00FF BF24", 1420, 240, -1);
994    text("0000 0008", 1420, 290, -1);
995    text("0000 6EBE", 1420, 340, -1);
996
997    //cmd
998    fill(255);
999    textFont(createFont("Arial", 30, true));
1000   text("Command Line", 970, 80);
1001   strokeWeight(3);
1002   stroke(color3);
1003   rect(900, 100, 350, 40, 10, 10, 10, 10);
1004   textFont(createFont("Arial", 26, true));
1005   fill(color4);
1006   text(typing, 900, 130);
1007   fill(255);
1008   rect(900, 350, 350, 500, 20, 20, 20, 20);
1009   fill(color3);
1010   textFont(createFont("Arial", 35, true));
1011   text("INFORMATION", 950, 400, -1);
1012   textFont(createFont("Arial", 21, true));
1013   text("Schedule is called by release.", 920, 460, -1);
1014   text("Procedure 4's counter is 6.", 920, 500, -1);
```

前半部分在绘制画布上原有的信息，包括 Command Line 命令行、information 信息框以及寄存器单元格序列。

```
1016    fill(stat1);
1017    n[0] = new node(300, 50, 200, 100);
1018    n[0].paint_trangle();
1019    fill(255);
1020    textFont(createFont("Arial", 35, true));
1021    text("pid = 0", 350, 110, -1);
1022    line(400, 150, 400, 200);
1023    fill(stat1);
1024    n[1] = new node(300, 200, 200, 100);
1025    n[1].paint_trangle();
1026    fill(255);
1027    textFont(createFont("Arial", 35, true));
1028    text("pid = 1", 350, 260, -1);
1029    line(400, 300, 200, 350);
1030    fill(stat1);
1031    n[3] = new node(100, 350, 200, 100);
1032    n[3].paint_trangle();
1033    //line(200, 350, 400, 300);
1034    fill(255);
1035    textFont(createFont("Arial", 35, true));
1036    text("pid = 3", 150, 410, -1);
1037    line(400, 300, 600, 350);
1038    fill(chosen);
1039    n[4] = new node(500, 350, 200, 100);
1040    n[4].paint_trangle();
1041    //line(200, 350, 400, 300);
1042    fill(255);
1043    textFont(createFont("Arial", 35, true));
1044    text("pid = 4", 550, 410, -1);
1045 }
1046
```

后半部分在更新进程树，这里是释放 5 进程的空间，所以在进程树上的体现就是重新绘制除 5 进程以外的其他部分进程树。如果是创建进程，就是把新的节点加入进程树；如果只是切换，就更新进程树中各个节点的颜色状态，在 information 信息框中显示当前的更新信息，在寄存器中更新压栈的值。

实验结果

数据提取部分结果

由于系统只要在工作，就不停地进行进程调度进程切换，就算已经没有额外的进程，也会在零进程 idle。故只要系统没有关闭，数据就一直在提取。除去最

终在零进程 idle 的数据，提取的数据大约有 8000 行；如果不除去最终执行零进程的时间，并且关闭系统及时，大约有两万行。（进程切换得很快，操作不迅速就会有很多执行零进程输出的数据）

格式如下：

```
Schedule is called by sleep_on.
Maybe 15 is the longest counter and 5 is the chosen one?
The chosen task:
4: pid=5, state=0, counter = 15, father=4, ss = 17, esp = 3fff0b4, cs = f, eip = 398af
jiffies is 13


all tasks as follows:
0: pid=0, state=1, counter = 14, father=-1, ss = 10, esp = 200d4, cs = 8, eip = 6ebe
jiffies is 13


1: pid=1, state=1, counter = 13, father=0, ss = 10, esp = ffff84, cs = 8, eip = 6ebe
jiffies is 13


2: pid=4, state=2, counter = 11, father=1, ss = 17, esp = 25e50, cs = f, eip = 6a5a
jiffies is 13


3: pid=3, state=1, counter = 14, father=1, ss = 10, esp = fc1f94, cs = 8, eip = 6ebe
jiffies is 14


4: pid=5, state=0, counter = 15, father=4, ss = 17, esp = 3fff0b4, cs = f, eip = 398af
jiffies is 14
```

最上部的信息是当前进程切换的原因。

第二行是进行进程调度算法的过程——从后往前遍历寻找最大的 counter，然后输出当前选中的进程的信息。

然后是当前被选中的进程的信息，包括进程的 pid，进程的状态，进程的优先级，进程的父进程，进程的寄存器信息和当前的 jiffies。

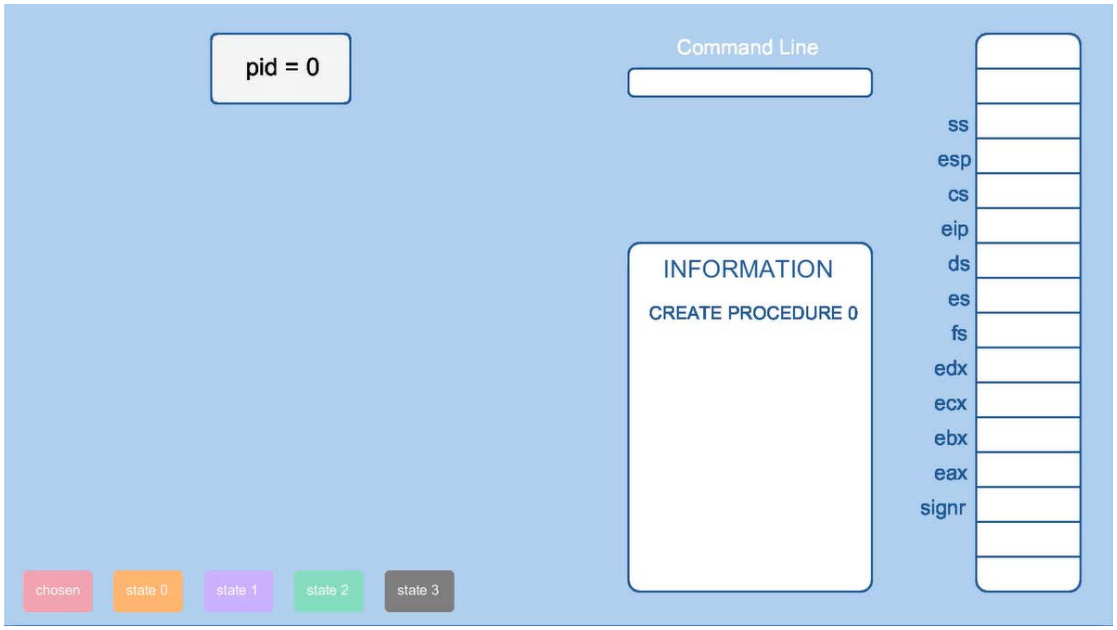
最后是当前所有进程的信息。

（完整数据见附件）

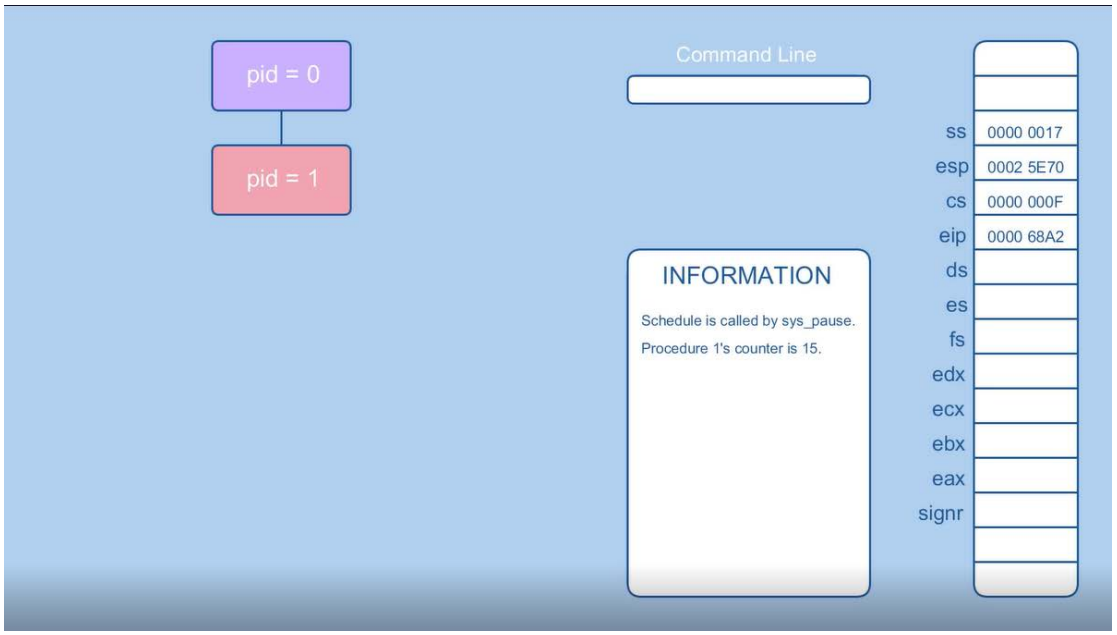
可视化部分结果

完成了进程创建、销毁在主界面上的可视化以及进程调度部分全部的可视化。个人工作量为 processing 代码 1868 行。并把小组三个人的代码组合起来，共代码 3276 行。对最后的视频做了剪辑和加入音频，使视频看起来更有趣。

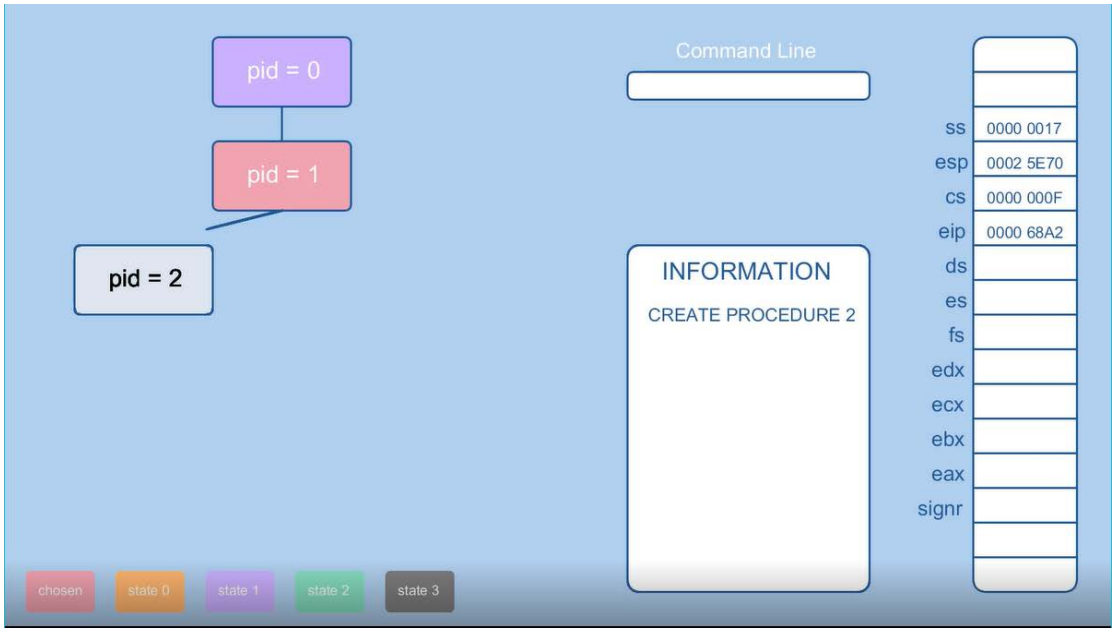
视频部分截图如下：



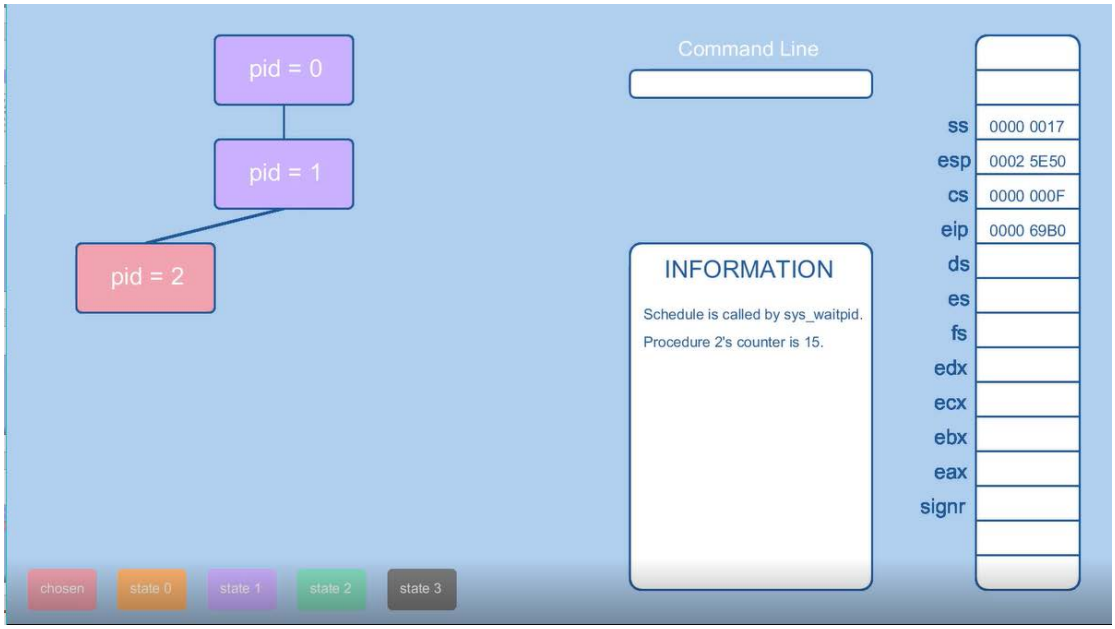
图为零进程被创建，并显示在进程树中



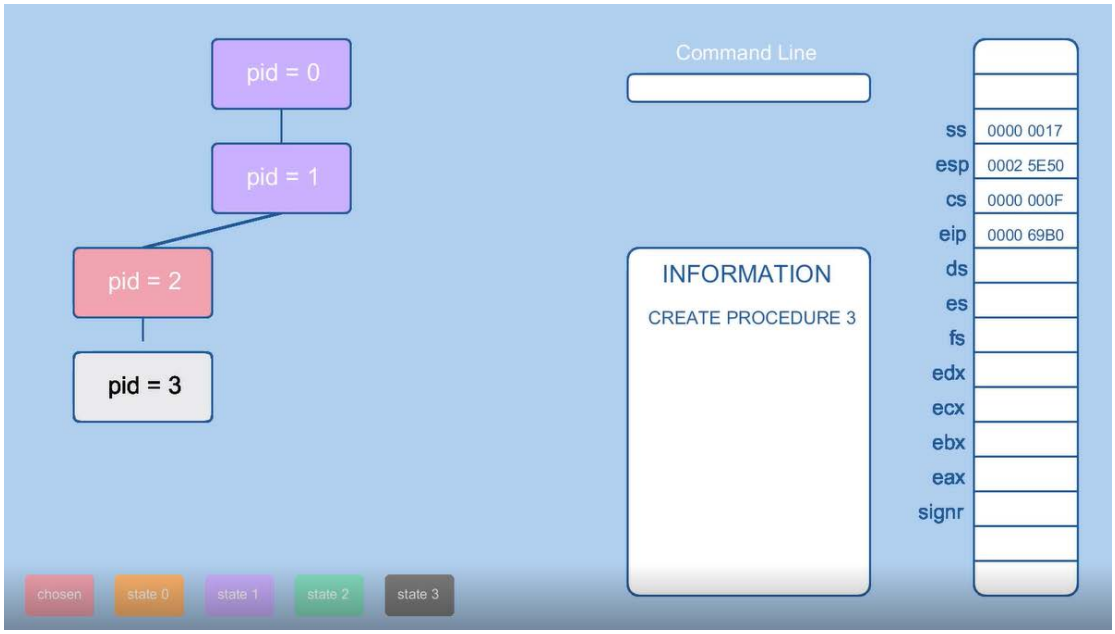
图为切换成 1 进程的状态，进程信息表里有按照算法选中的 1 进程的优先级，以及此时进程切换的原因。寄存器信息中显示压入栈中的信息。



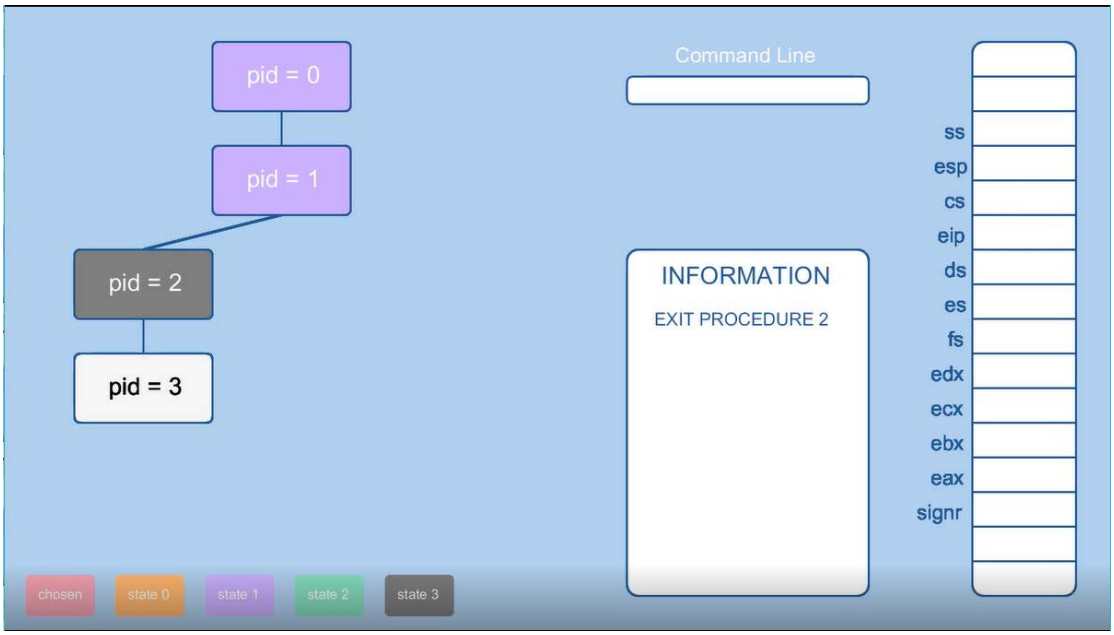
以 1 进程为父进程创建子进程 2 进程，图为正在创建并加入进程树的过程。



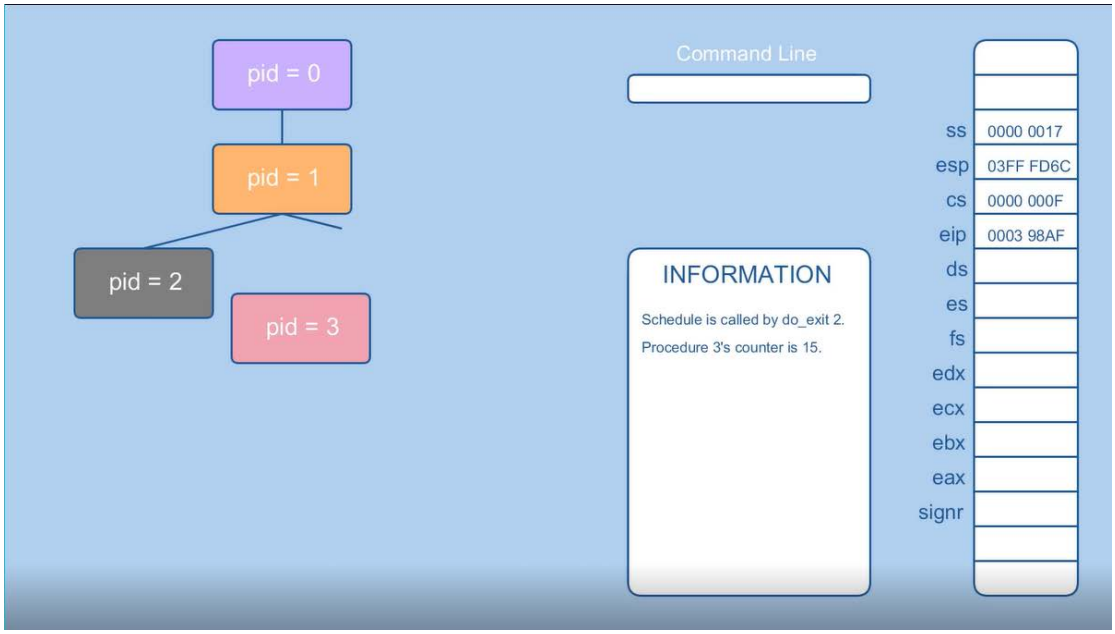
根据切换算法选择至优先级最高的 2 进程，并显示寄存器中信息



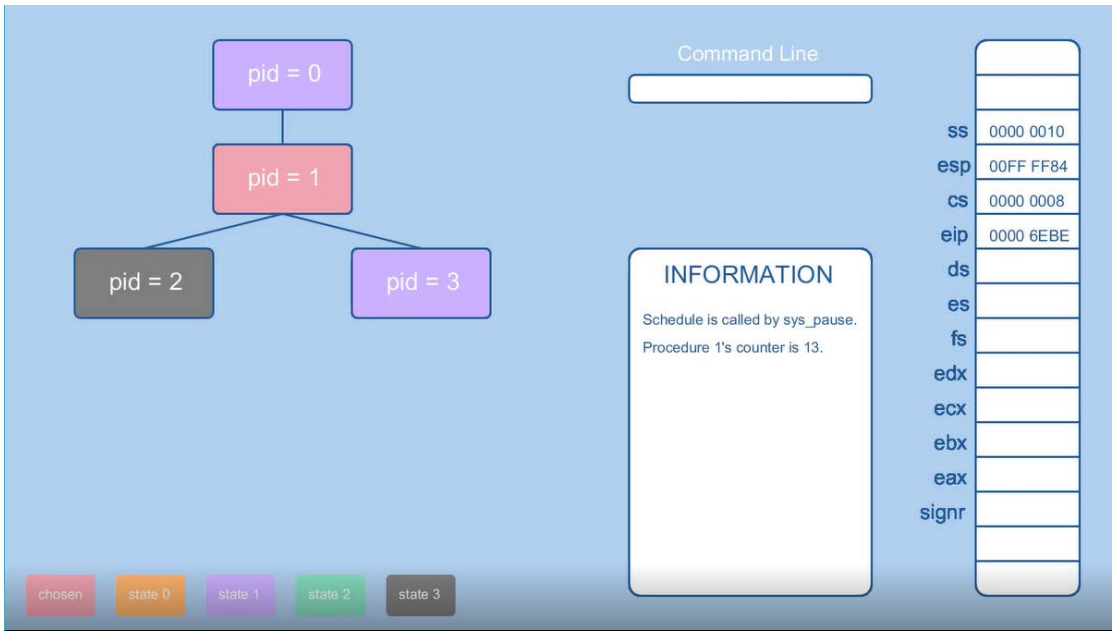
以 2 进程为父进程创建子进程 3 进程，图为正在创建并加入进程树的过程。



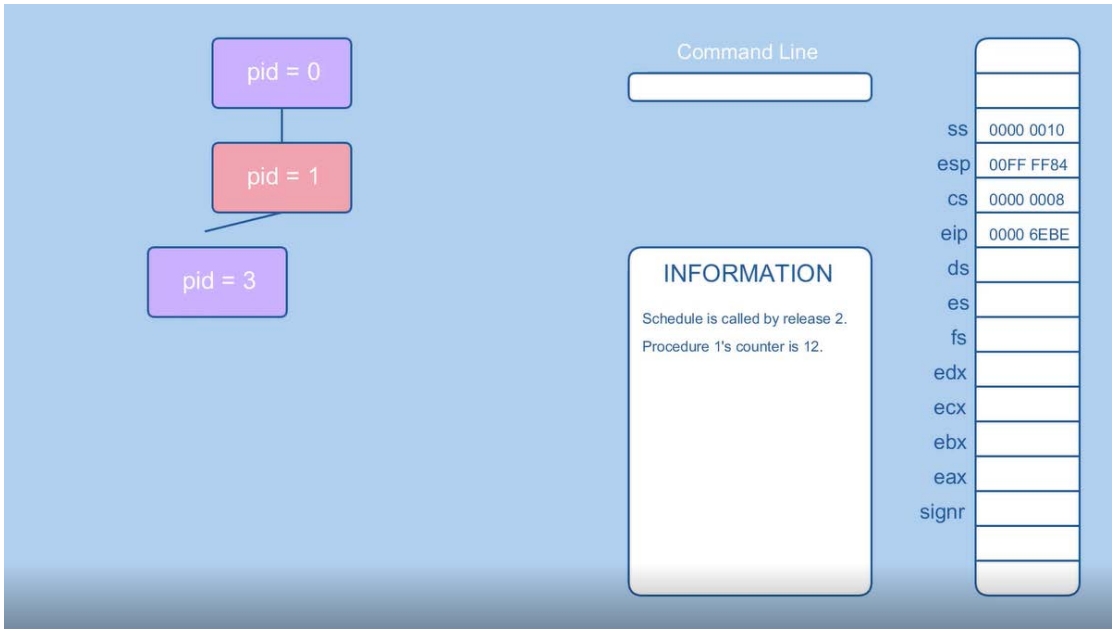
进程切换，把 2 进程从被选中的状态变为僵尸状态



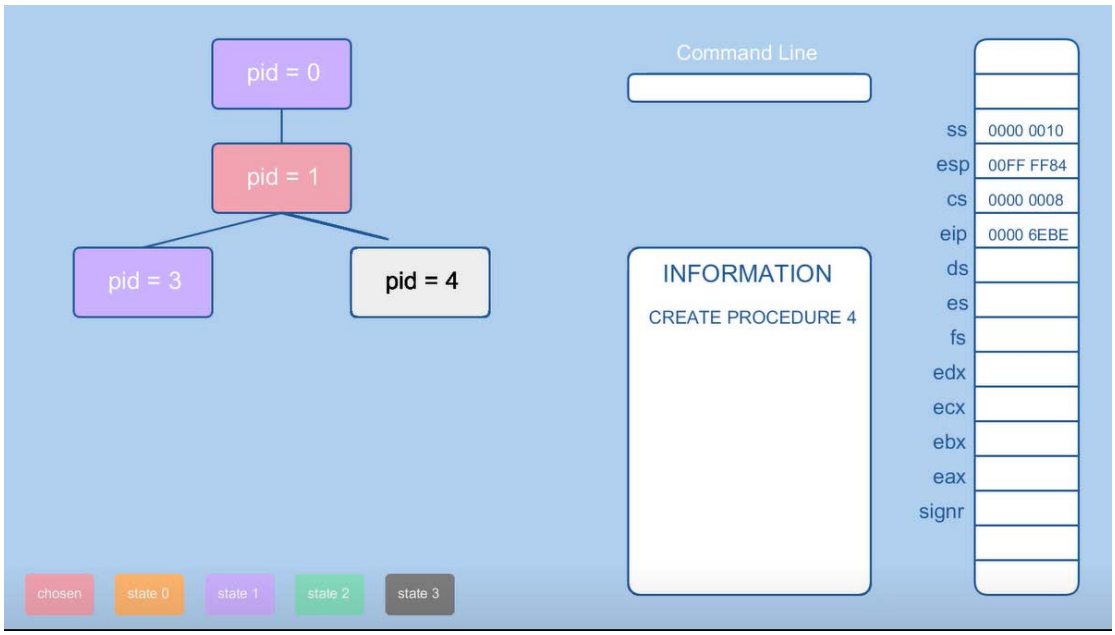
选中 3 进程，并把 3 进程的父亲变为 1 进程



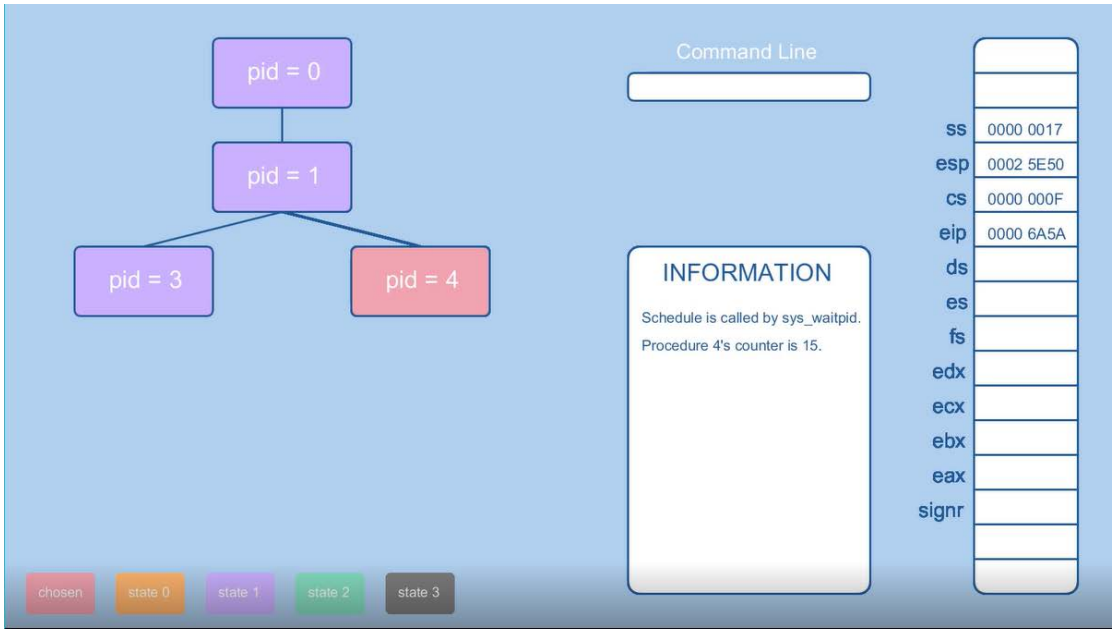
选中 1 进程，为 2 进程空间回收做准备



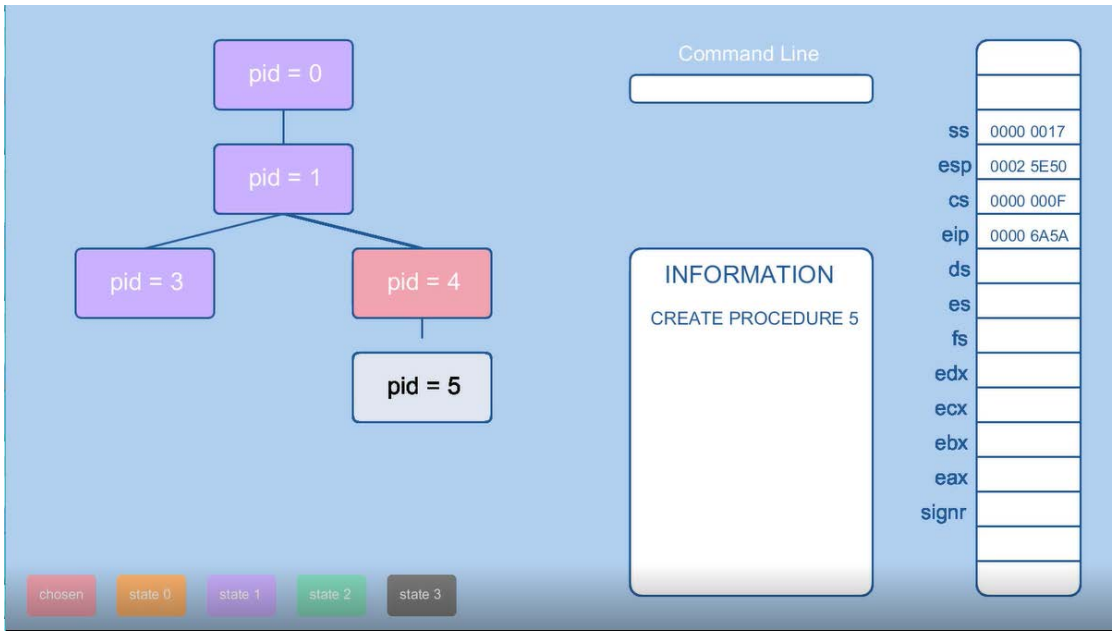
回收 2 进程空间



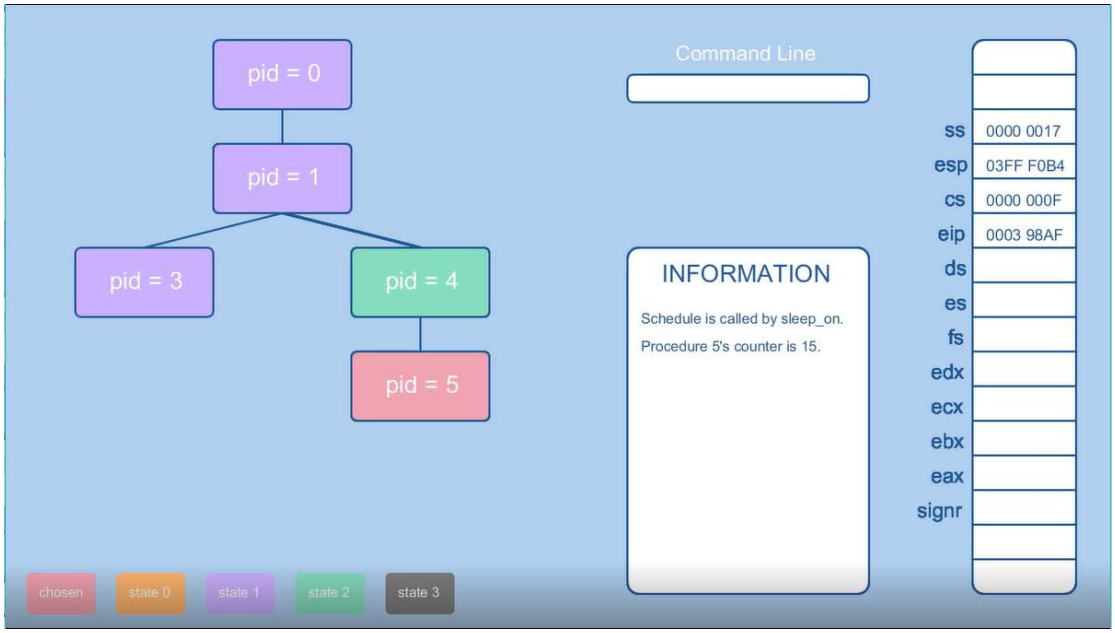
以 1 进程为父进程创建子进程 4 进程，图为正在创建并加入进程树的过程。



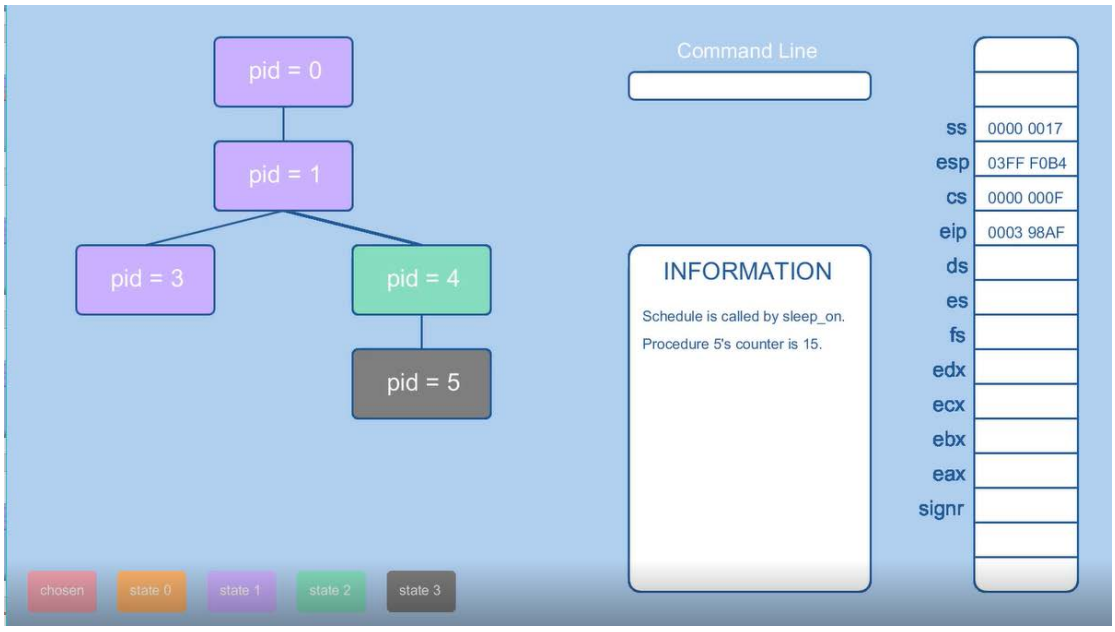
根据切换算法选择至优先级最高的 4 进程，并显示寄存器中信息



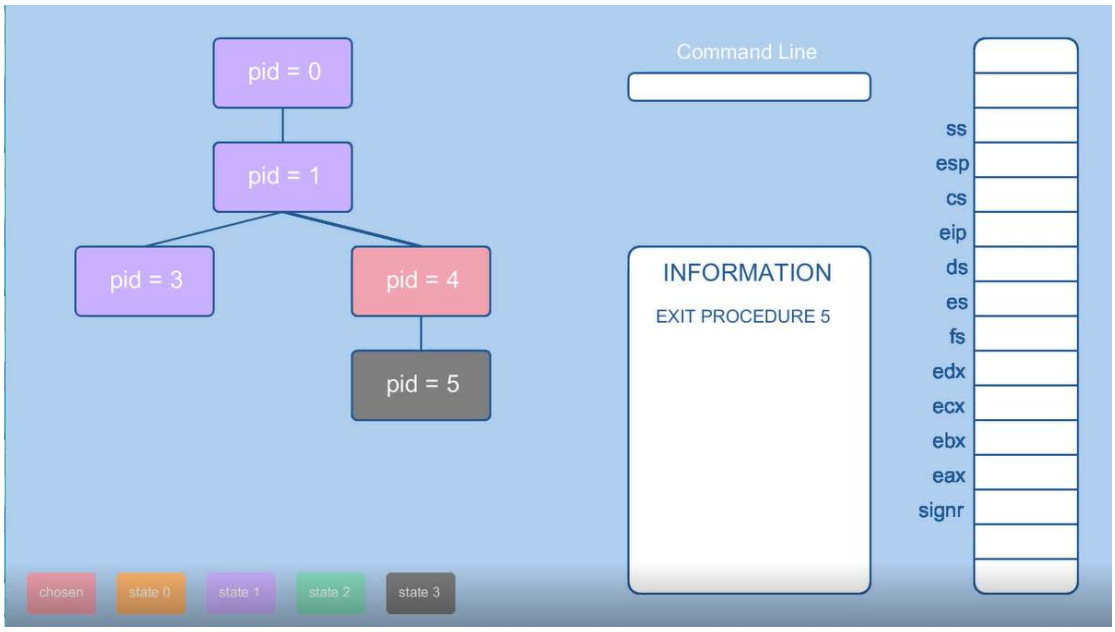
以 4 进程为父进程创建子进程 5 进程，图为正在创建并加入进程树的过程。



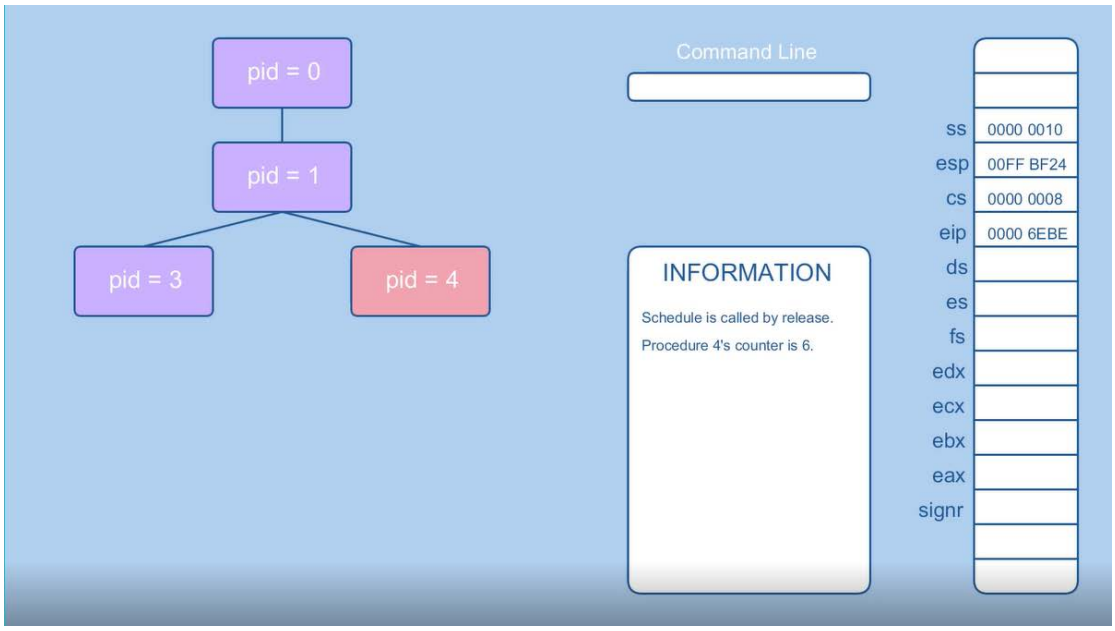
根据切换算法选择至优先级最高的 5 进程，将 4 进程设置成不可中断等待状态并显示切换原因以及寄存器中信息



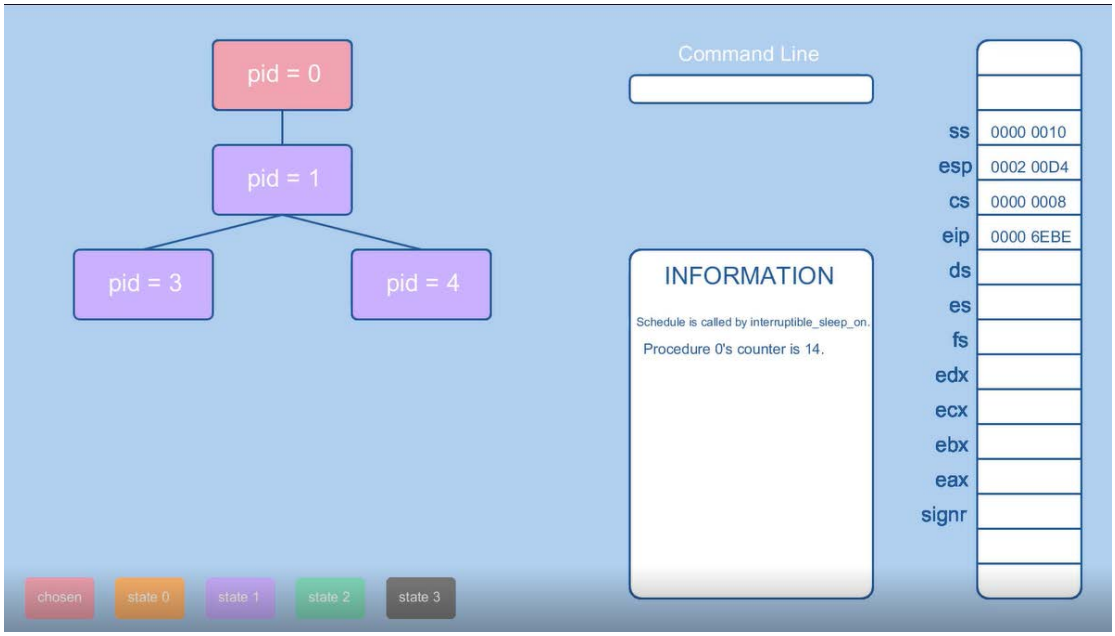
进程切换，把 5 进程从被选中的状态变为僵尸状态



选中 4 进程，为 5 进程空间回收做准备

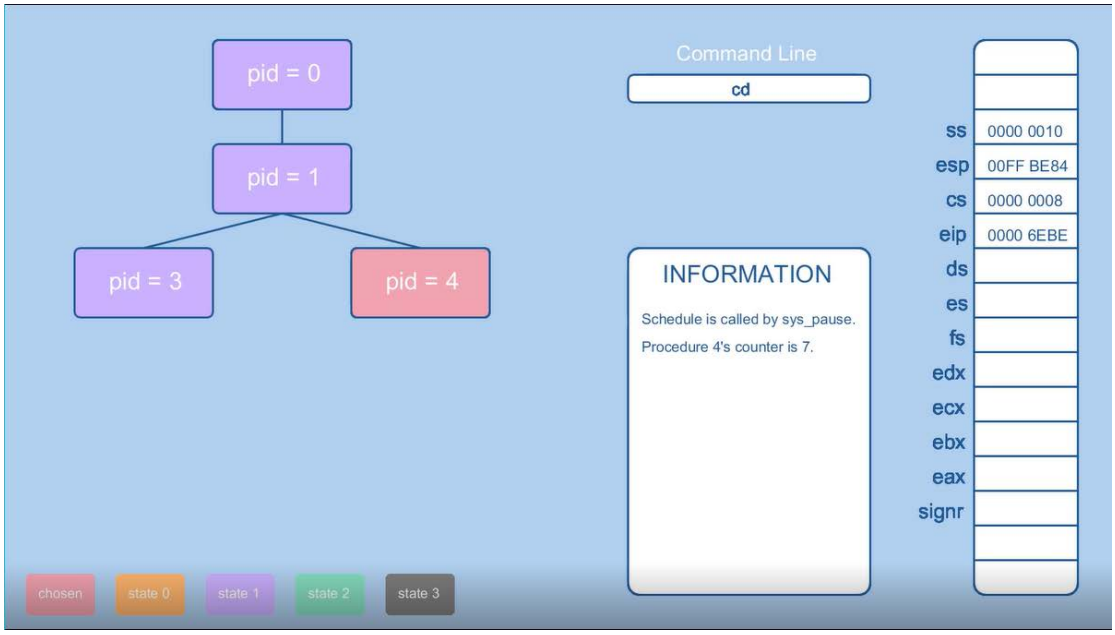


回收 5 进程空间

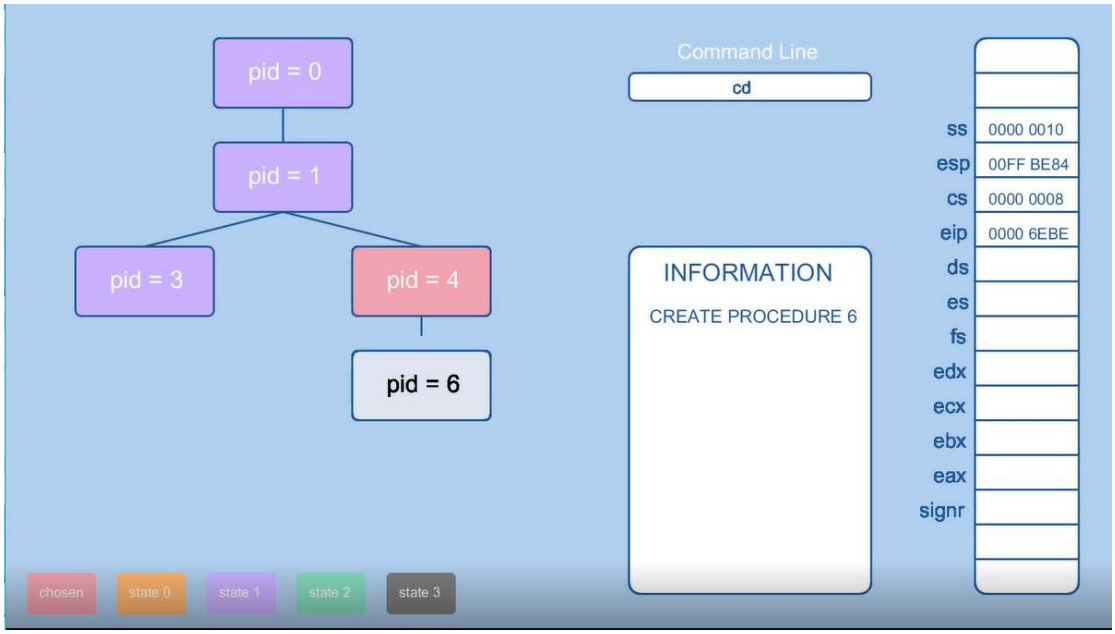


此时系统开机时的所有进程已经执行完毕，选中 0 进程进入 idle 状态，等待有的进程加入

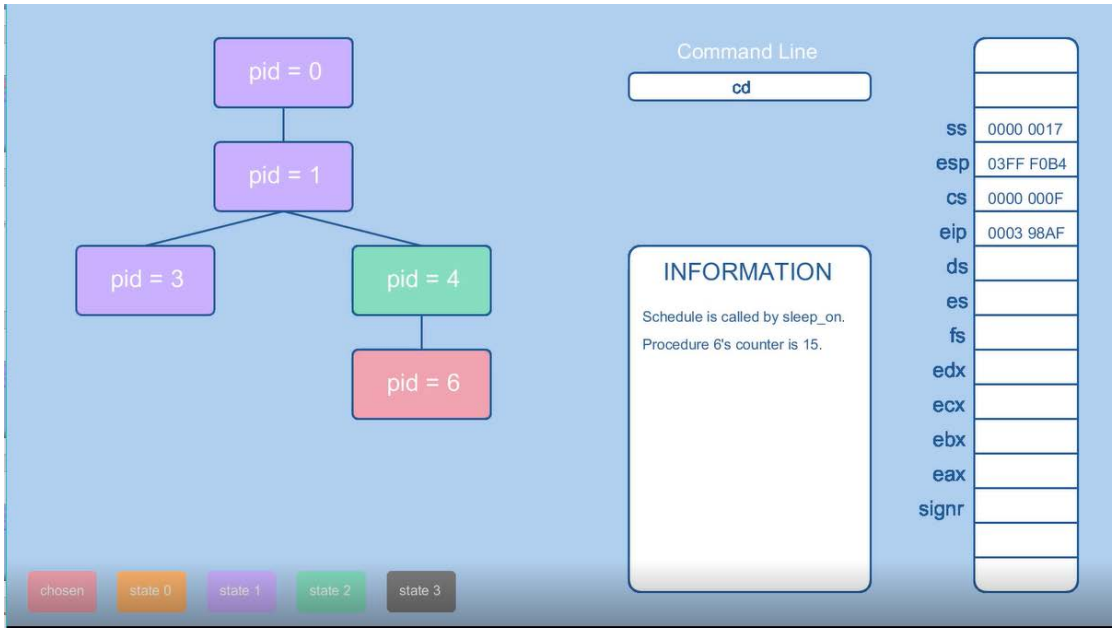
创建新的进程，在 qemu 中输出指令 cd、ls，cd 可以创建一个进程，ls 可以创建两个进程。



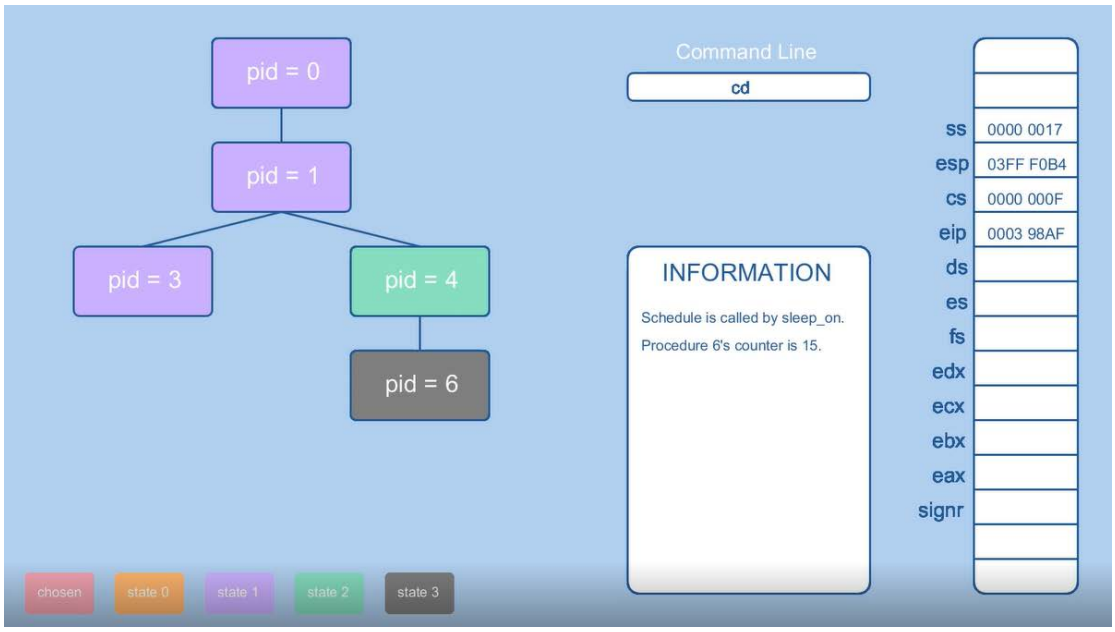
输出 cd，即将创建以 4 进程为父进程创建新的进程，选中 4 进程，并输出相关信息。



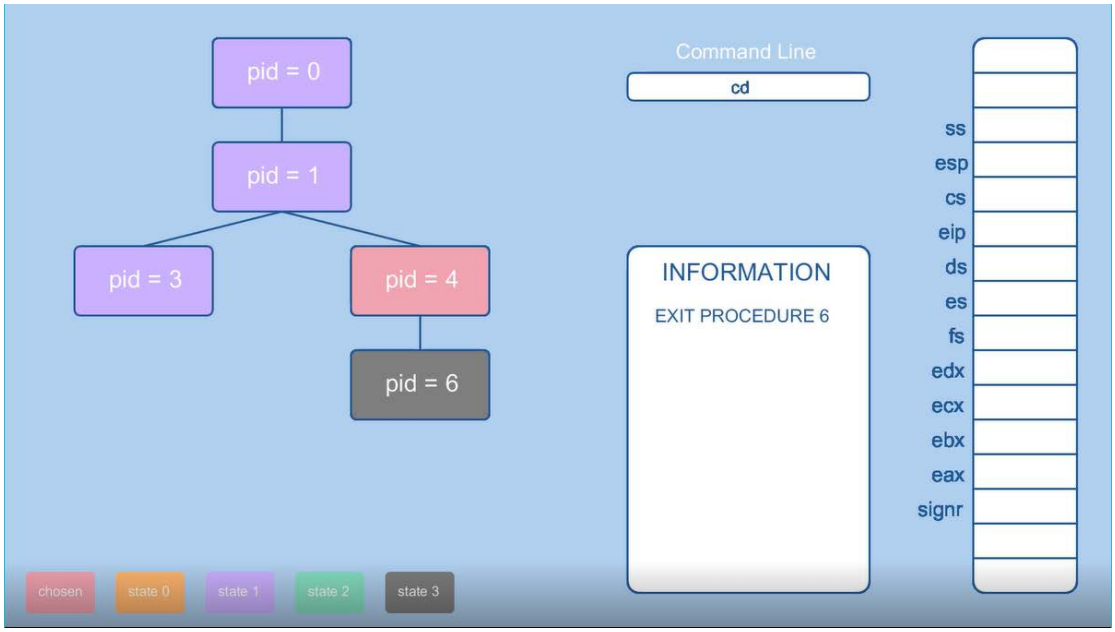
以 4 进程为父进程创建子进程 6 进程，及 cd 命令创建的进程，图为正在创建并加入进程树的过程。



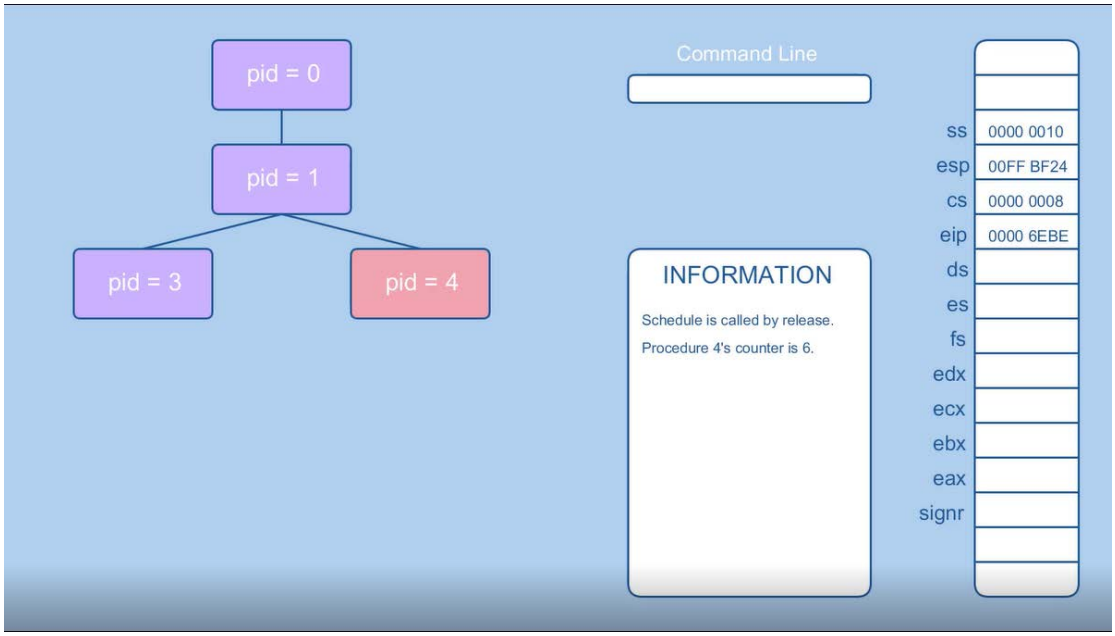
根据切换算法选择至优先级最高的 6 进程，4 进程设置成不可中断等待状态，并显示切换原因以及寄存器中信息



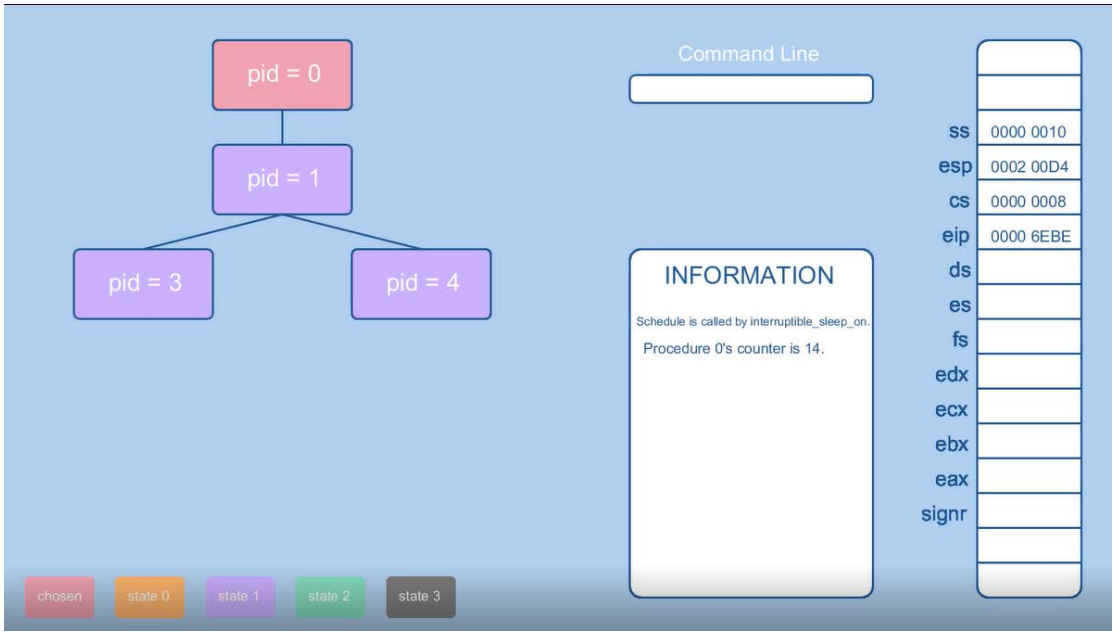
执行完 cd 后，把 6 进程从被选中的状态变为僵尸状态



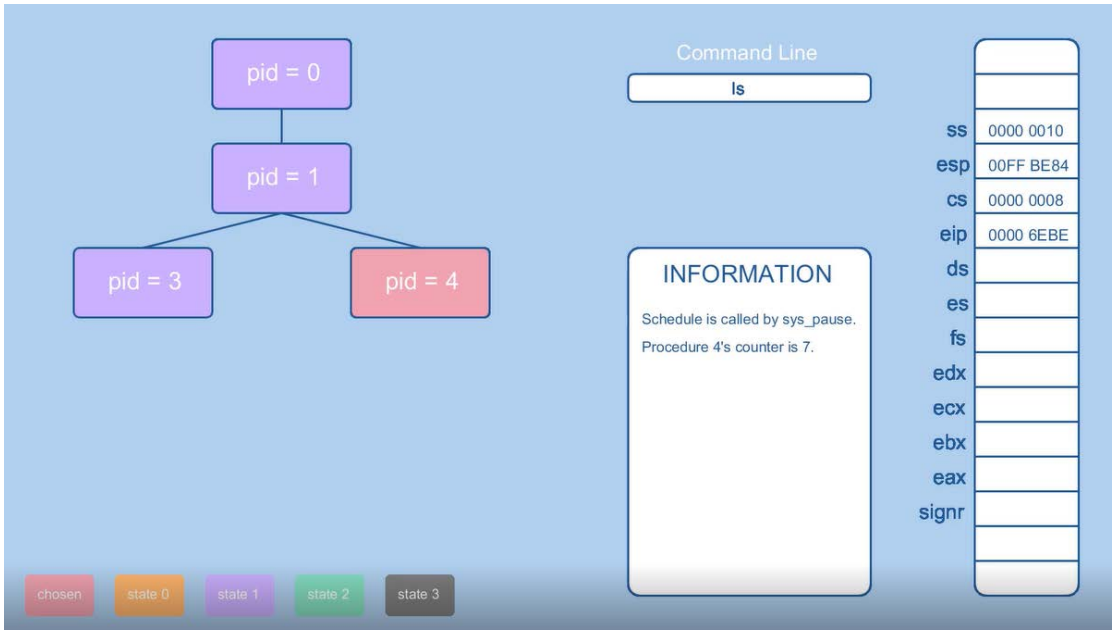
选中 4 进程，为 6 进程空间回收做准备



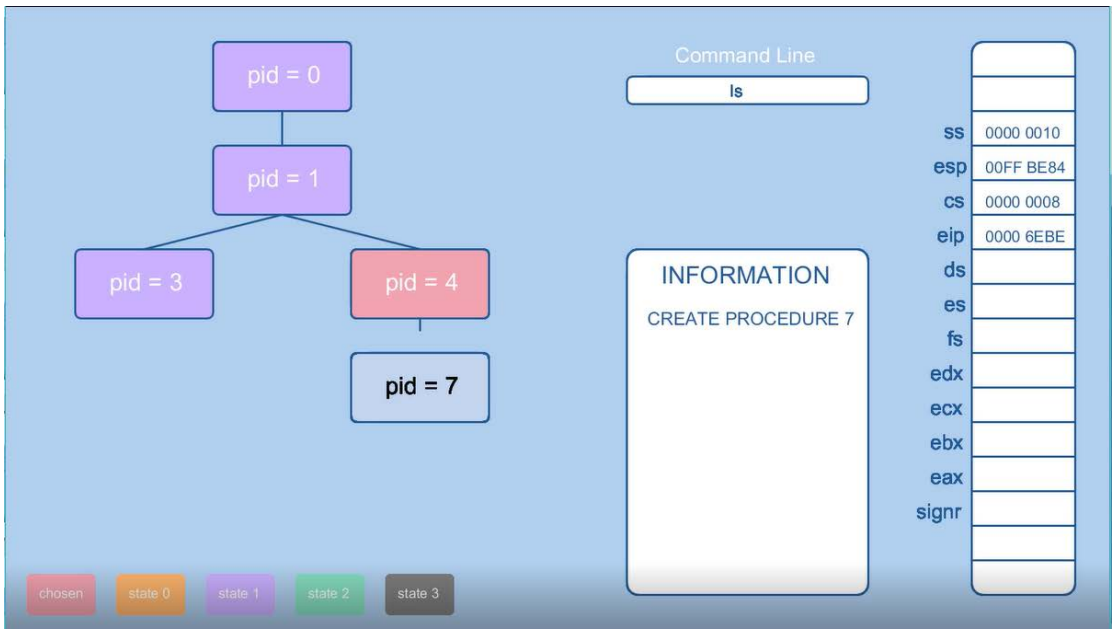
回收 6 进程空间



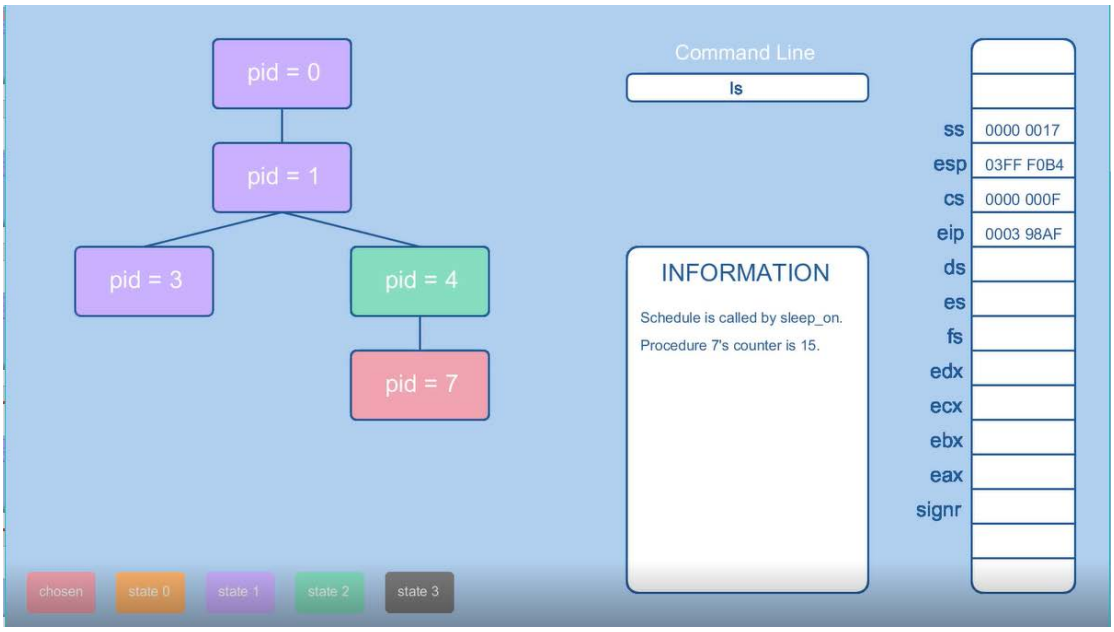
执行完毕 cd 的所有进程，再次选中 0 进程进入 idle 状态，等待有新的进程加入



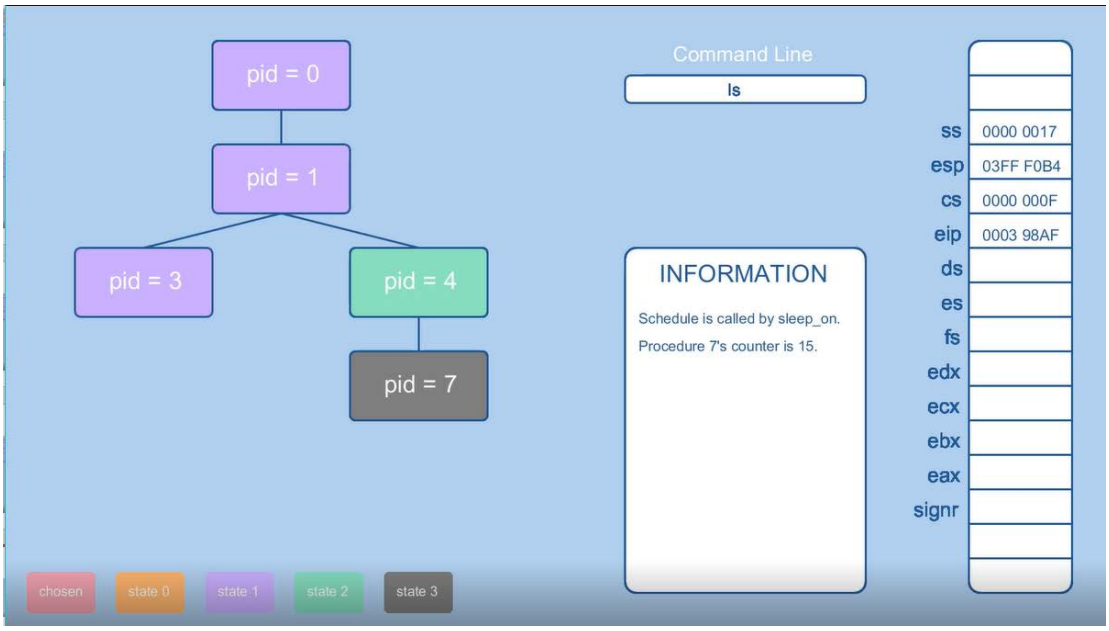
输出 ls，即将创建以 4 进程为父进程创建新的进程，选中 4 进程，并输出相关信息。

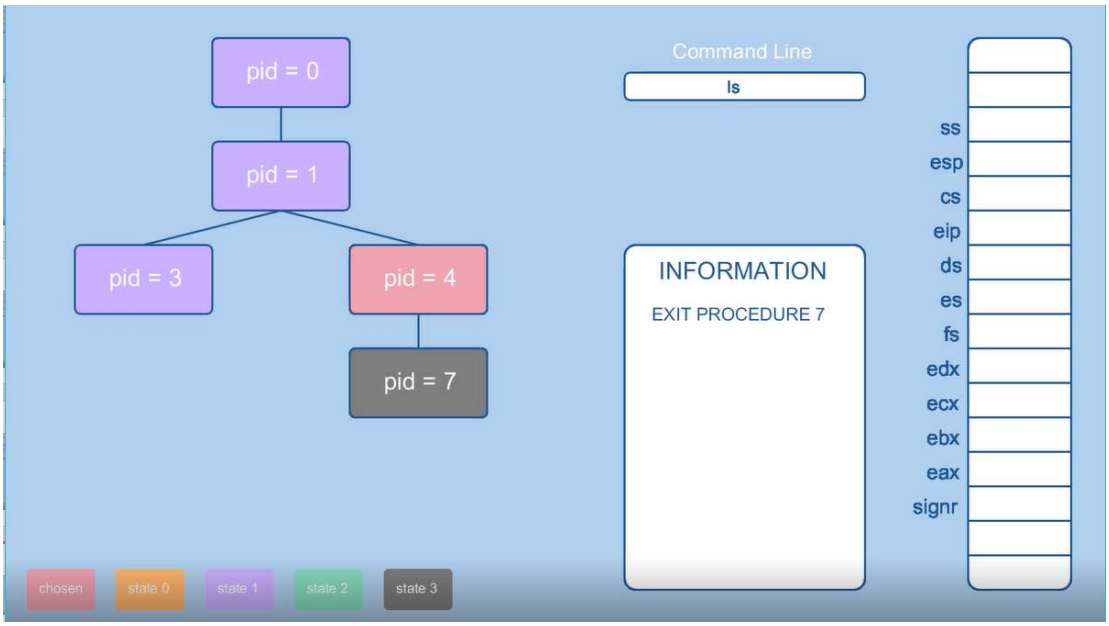


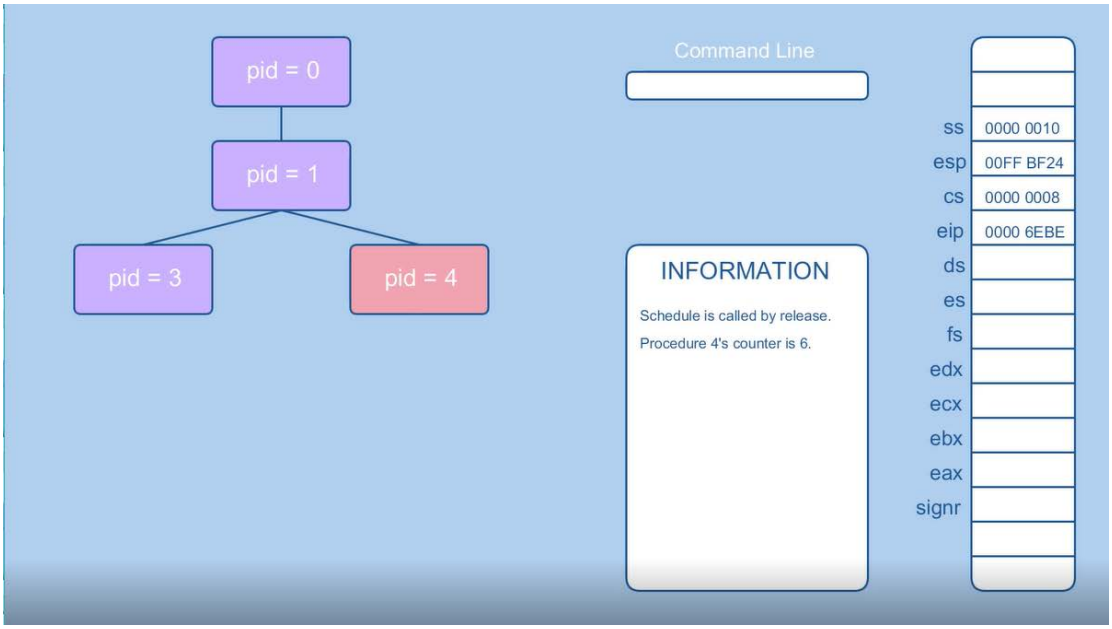
创建 ls 的第一个进程——7 进程



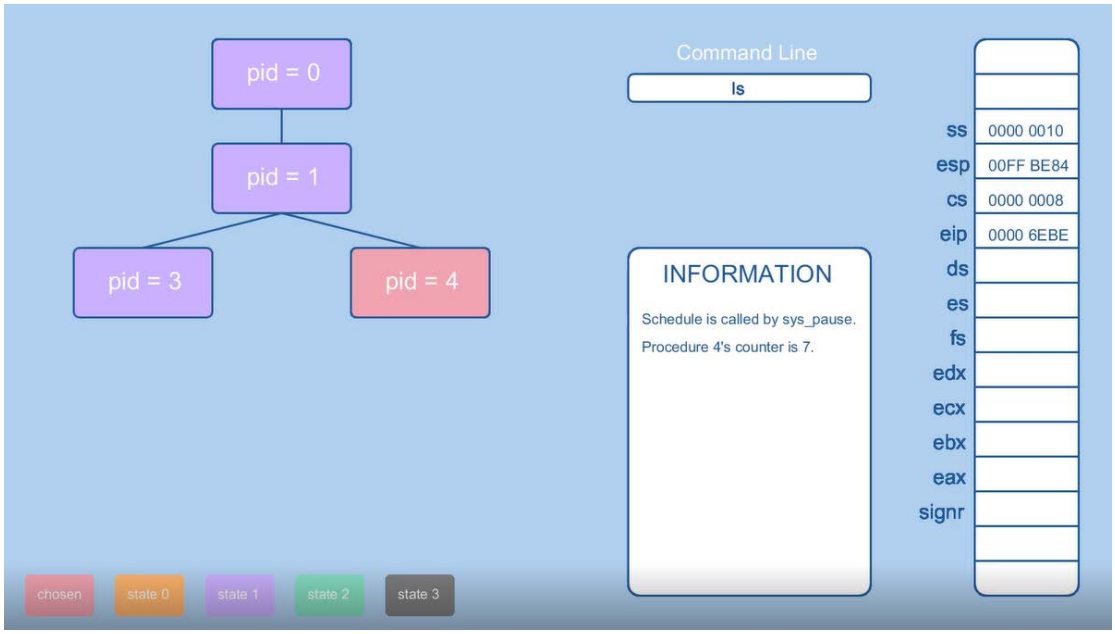
根据切换算法选择至优先级最高的 7 进程，将 4 进程设置成不可中断等待状态，并显示切换原因以及寄存器中信息。



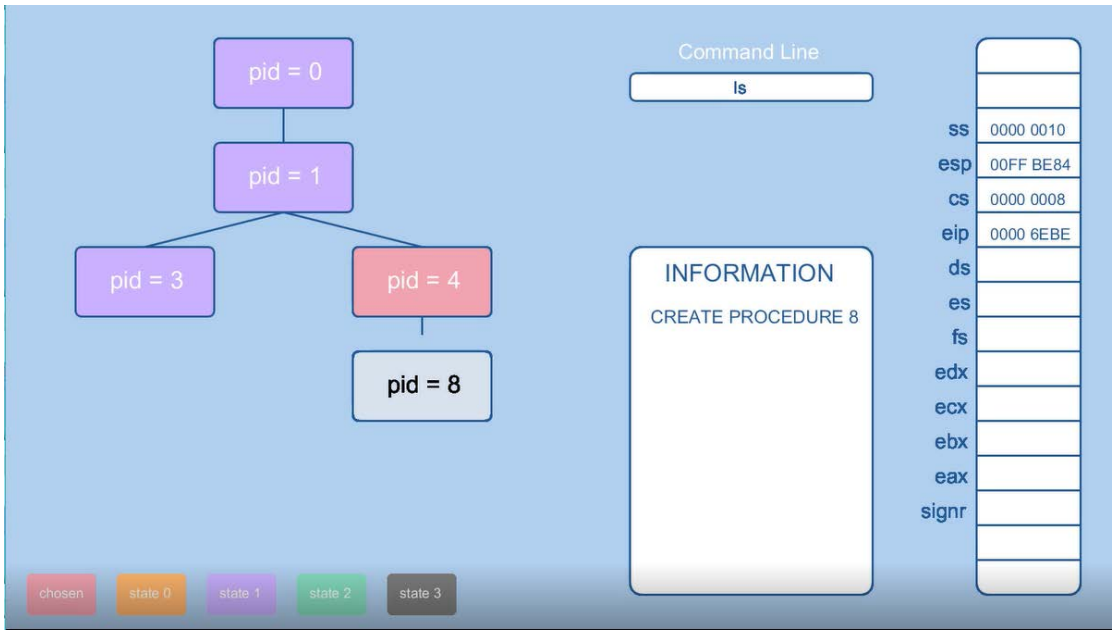

pid = 0
pid = 1
pid = 3
pid = 4
pid = 7
Command Line
ls
INFORMATION
EXIT PROCEDURE 7
ss
esp
cs
eip
ds
es
fs
edx
ecx
ebx
eax
signr
chosen
state 0
state 1
state 2
state 3


pid = 0
pid = 1
pid = 3
pid = 4
Command Line
INFORMATION
Schedule is called by release.
Procedure 4's counter is 6.
ss 0000 0010
esp 00FF BF24
cs 0000 0008
eip 0000 6EBE
ds
es
fs
edx
ecx
ebx
eax
signr

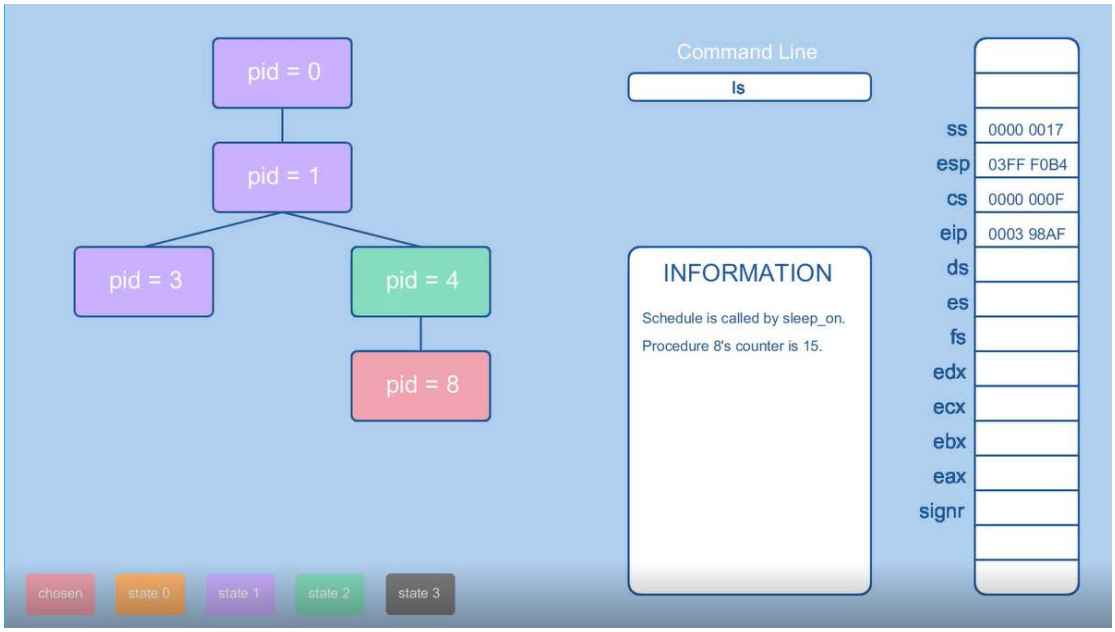
销毁回收 7 进程



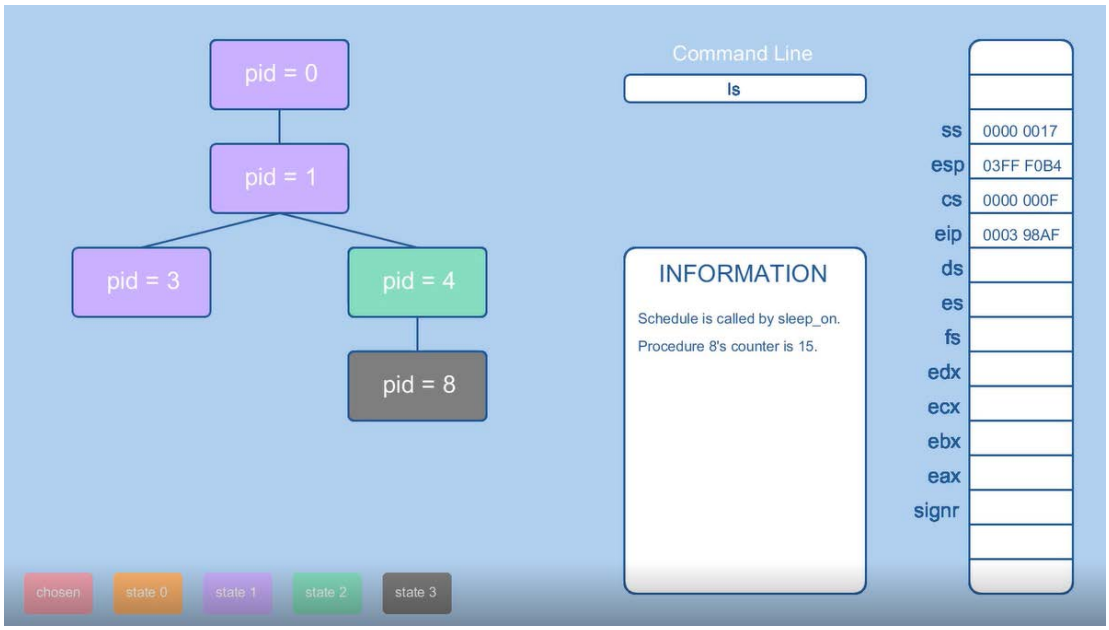
选中 4 进程，准备创建 ls 的第二个进程 8 进程，并输出相关信息

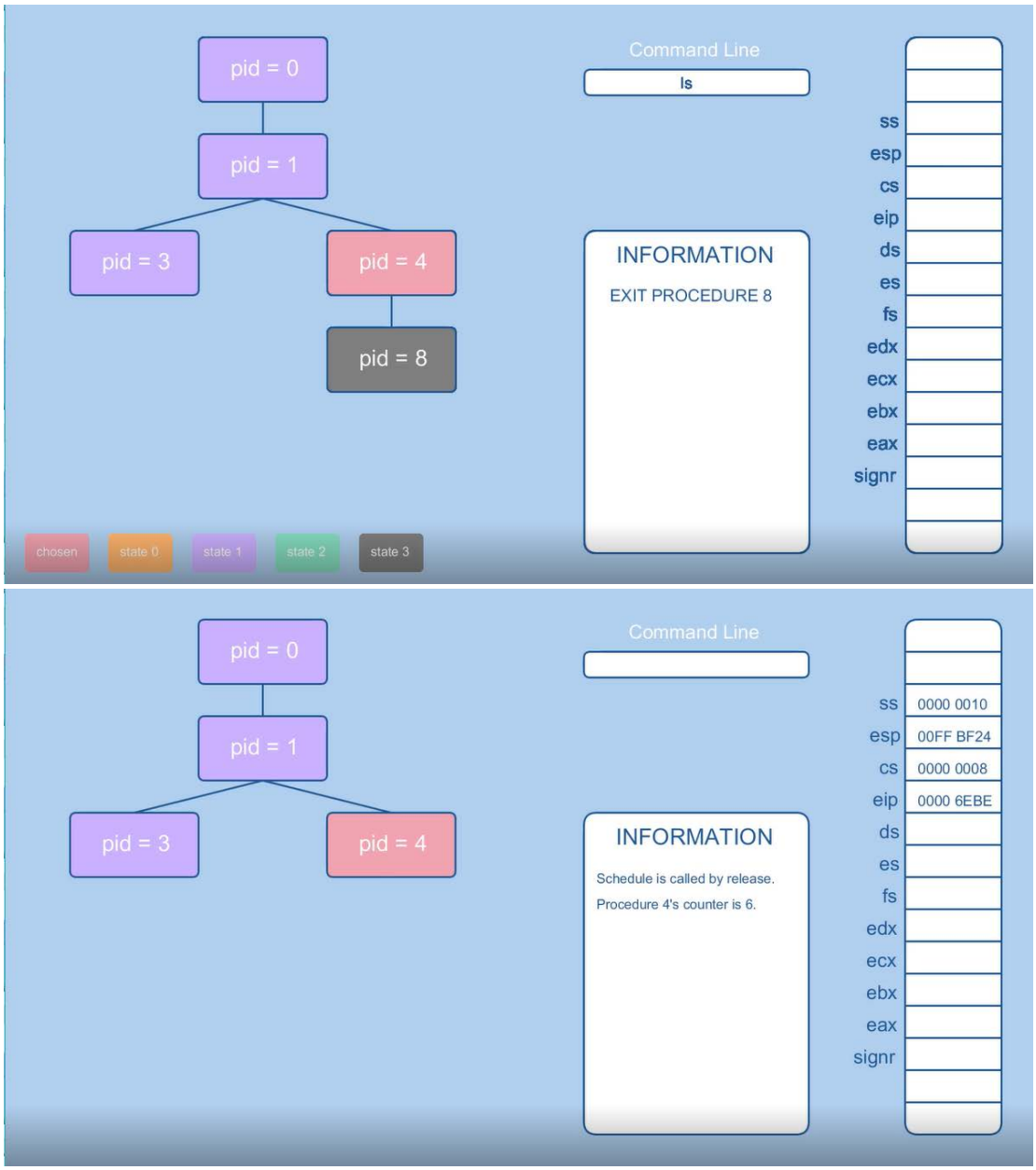


创建 ls 的第二个进程——8 进程



根据切换算法选择至优先级最高的 8 进程，将 4 进程设置成不可中断等待状态，并显示切换原因以及寄存器中信息





销毁回收 8 进程

（完整视频见附件）

问题及收获

问题

小组合作时要统一系统版本，我们小组合起进程创建与进程切换的数据时发

现版本不统一，输出的数据不一样，统一版本后又重新输出了一遍。

要尝试多种可视化工具，小组本来选择使用 PyQt 来可视化，但是由于与系统交互比较困难，所以 PyQt 失去了优势，重新选择了比较直观、方便做成视频展示的 processing。通过多种可视化工具的对比，可以选择最适合的，做出最想要的效果。

最后我的工作还是存在着不足，比如必须先提取数据然后可视化，不能实现与系统的交互，系统一边运行一边输出数据一边可视化。而且代码复用性较差。

收获

对 Linux 0.11 操作系统有了更加深刻的认识，理解了进程的初始化、创建、调度、销毁和通信的内容，提高了阅读源码的能力，学会使用 processing，掌握了如何编辑视频，在与同组成员的交流中。增强了小组合作的意识。