在本部分中，我们将要动手实现一个多级反馈队列的调度算法，其需要满足 符合立即抢占，即如果有一个新的优先级高的进程加入进来，当前正在运行的优先级较低的进程会被立即剥夺执行权限进入休眠状态，被剥夺的进程仍然放到当前队列中；

本次实验也有一定的局限性：实验中所有用到的进程都是我们实现创建好的，我们无法在运行时动态地随机创建一个进程，所以抢占与剥夺的效果在本实验中体现得并不明显。

本实验的整体思路如下：

仍然使用上次实验中基于时钟中断的定时器，每次定时器一到时间，和教材中给出的代码一样，遍历proc\_table找出最高优先级的进程并赋予执行权限；

对于多级反馈队列的时间问题，我们规定一个基本的时间单位T = n次时钟中断，然后各级队列的时间片长度分别为T、2T、4T ……

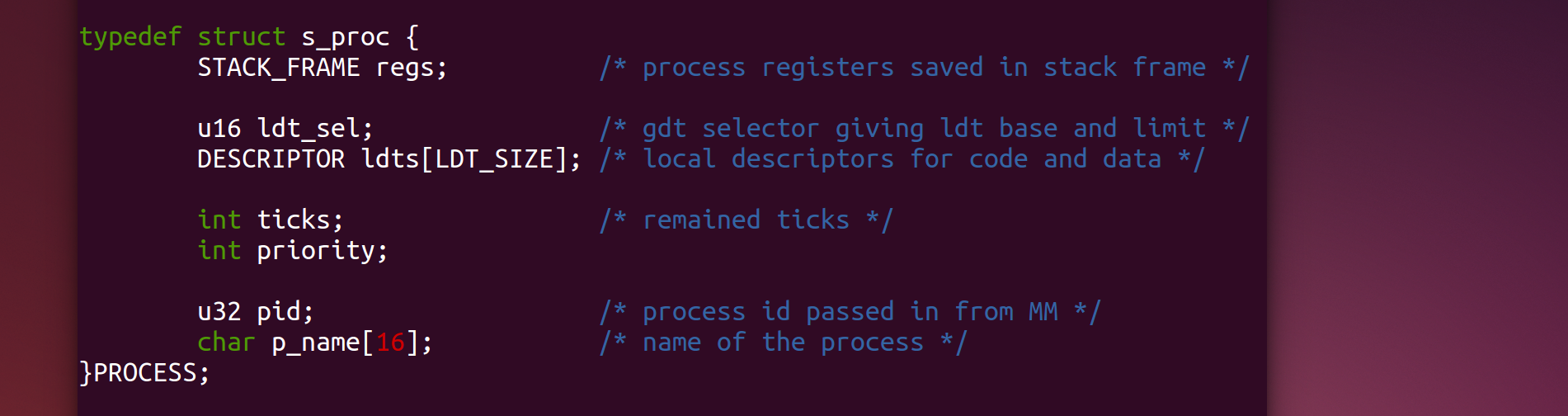
关键要解决以下两个问题：

1. 如果一个进程被终止，那么如何如何记录该进程的时间片？

所以我们先实现相关的数据结构：

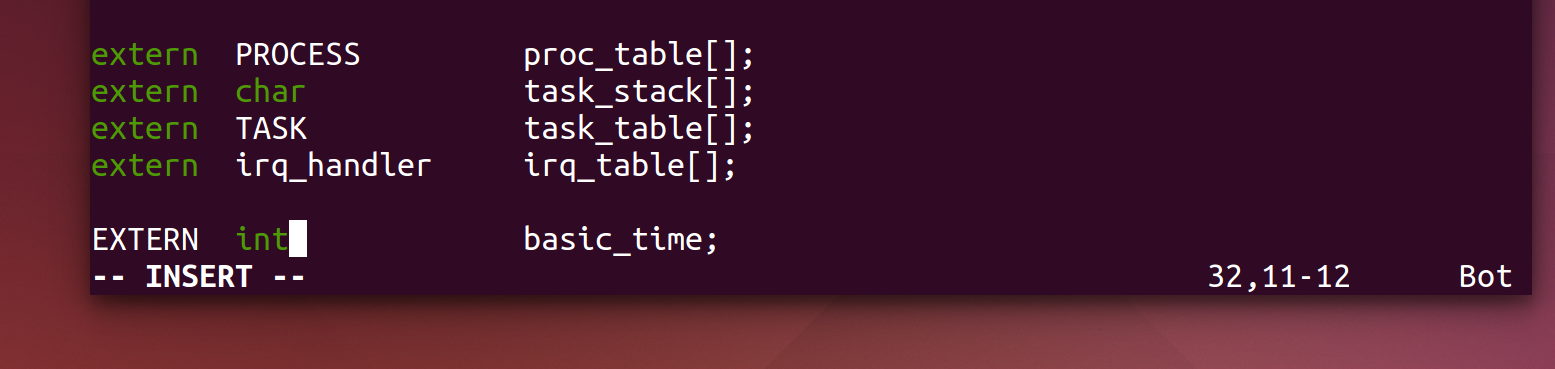
1. 用一个变量记录该进程属于哪一个优先级队列，然后再用一个变量来记录该进程在当前队列中的剩余时间；

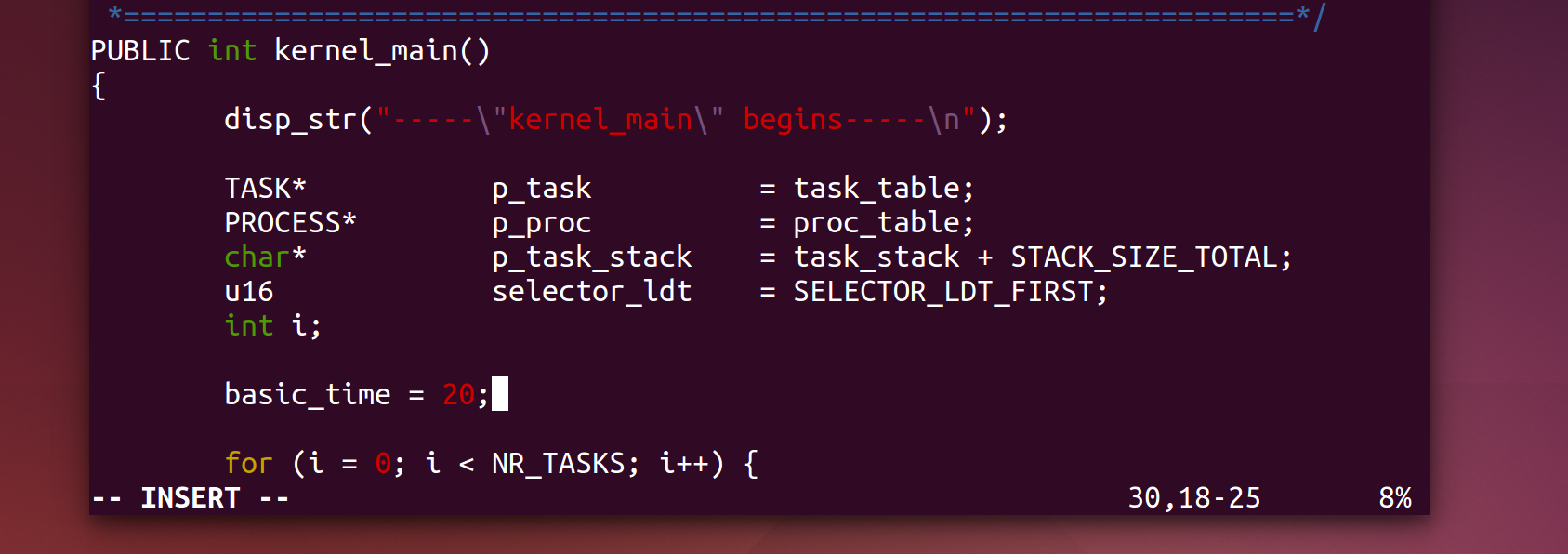
经过分析，我们仍然可以使用/include/proc.h中的PROCESS结构体：



我们使用priority变量来描述进程属于哪一个优先级队列，ticks变量表示该进程在当前队列中剩余的时间；

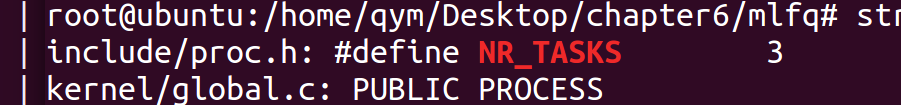
在这里我们定义时间片基本单位T = 20，在global.h中定义一个全局变量basic\_time，并在kernel\_main中进行初始化：

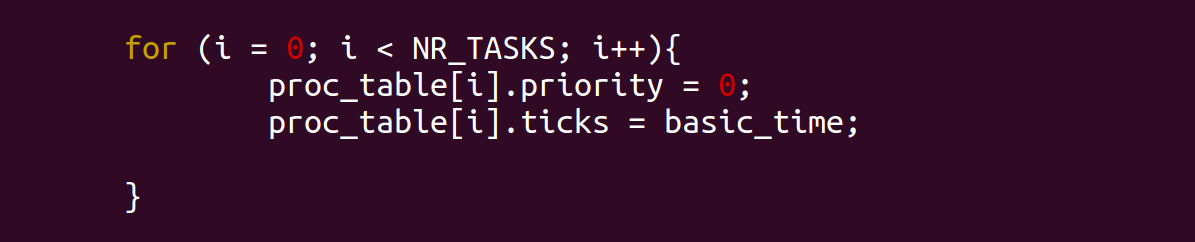




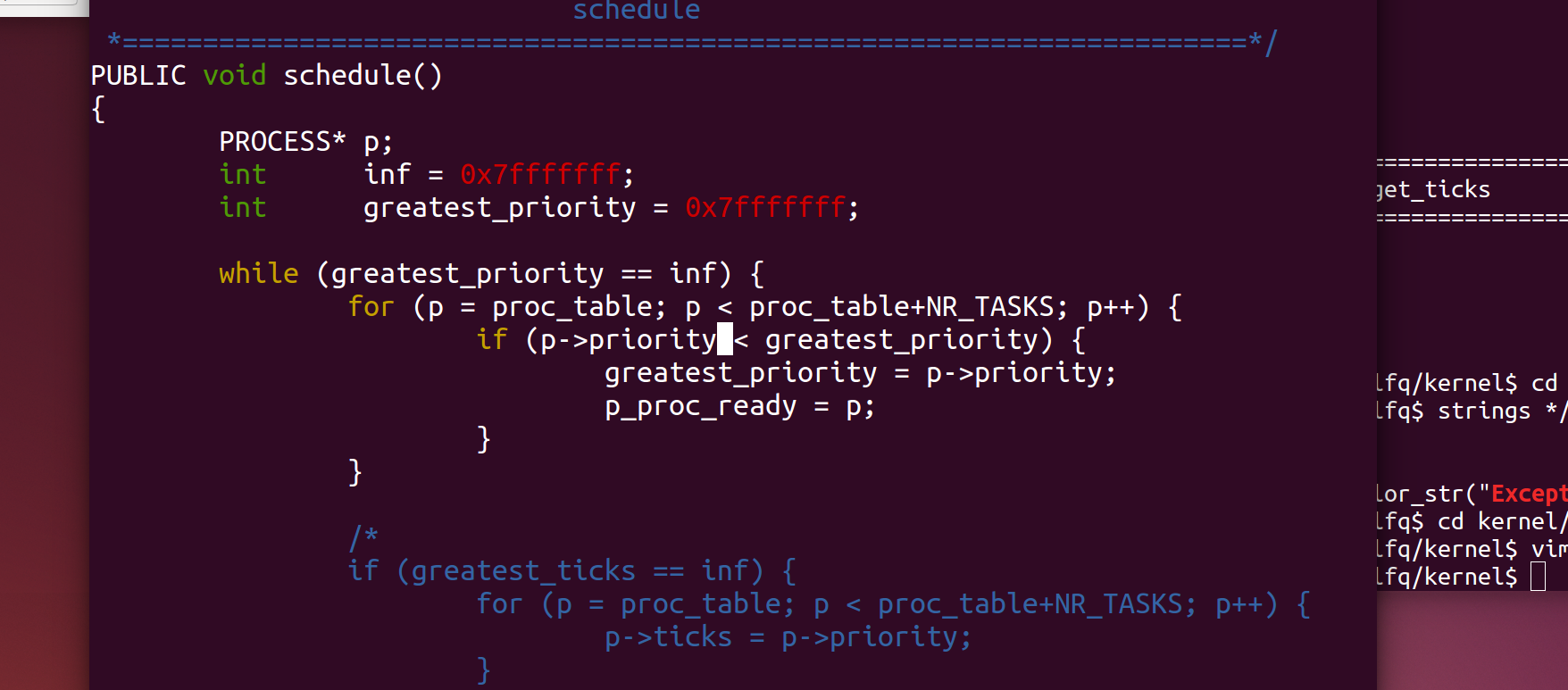
多级调度中我们使用三级队列，且规定级别越高priority的值越小（最高级位0），在初始化时，我们将所有进程的priority赋值为0，均处于最高级队列；

对于ticks的值，则都是以T为单位的，在main函数里初始化每一个进程的ticks：





由于我们采用priority越小优先级越高，所以我们改变一下schedule中的逻辑：



最终我们的时钟中断处理程序如下：

PUBLIC **void** clock\_handler(**int** irq)

{

    ticks++;

    p\_proc\_ready->ticks--;

    PROCESS\* p;

**int** have\_high;

    if (k\_reenter != 0) {

        return;

    }

    if(p\_proc\_ready->priority == 0){

*/\*如果当前进程在最高优先级队列且时间片未用完，则一定不会被抢占\*/*

            if(p\_proc\_ready->ticks != 0) return ;

*/\*时间片用完\*/*

        p\_proc\_ready->ticks = basic\_time \* 2;

        p\_proc\_ready->priority = 1;

        schedule();

    }

    else if(p\_proc\_ready->priority == 1){

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

            p\_proc\_ready->ticks = basic\_time \* 3;

            p\_proc\_ready->priority = 2;

            schedule();

            return ;

        }

        have\_high = 0;

*/\*检查是否有优先级更高的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(p->priority == 0){

                have\_high = 1;

                break;

            }

        }

        if(have\_high){

*/\*如果有优先级更高的进程\*/*

*/\*则发生抢占\*/*

            p\_proc\_ready = p;

        }

        }

    else if(p\_proc\_ready->priority == 2){

**int** have0 = 0;

**int** have1 = 0;

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

*/\*由于进程在最低优先级队列，不会降级，仅恢复时间片\*/*

                        p\_proc\_ready->ticks = basic\_time \* 3;

                        schedule();

                        return ;

                }

*/\*检查是否有优先级为0的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

                        if(p->priority == 0){

                                have0 = 1;

                                break;

                        }

                }

*/\*检查是否有优先级为1的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(have0) break;

            if(p->priority == 1){

                have1 = 1;

                break;

            }

        }

        if(have0 || have1){

*/\*如果有更高优先级的进程就发生抢占\*/*

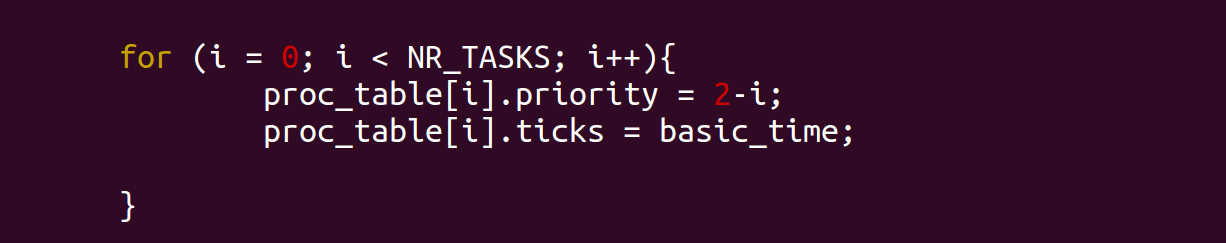
            p\_proc\_ready = p;

        }

        }

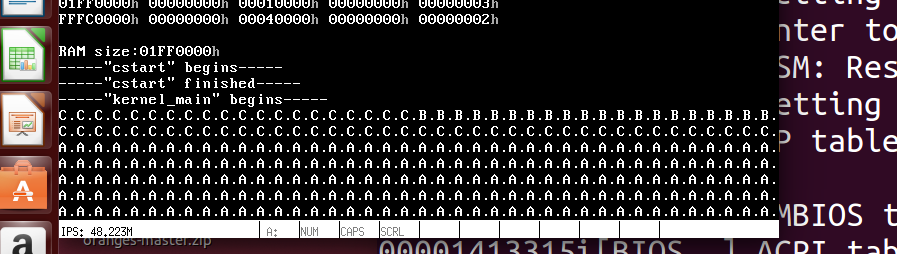
}

下面我们将对我们的调度算法进行测试，为了方便展示效果，我们将三个进程的优先级实现倒序，即C > B > A:



这样，预期效果应该是C优先级最高先打印若干C，之后C时间片用完，CB同时处于优先级1，则先打印B再打印C，之后ABC同时处于优先级2，则A会一直在前面，则会一直打印A；

执行之后看效果：

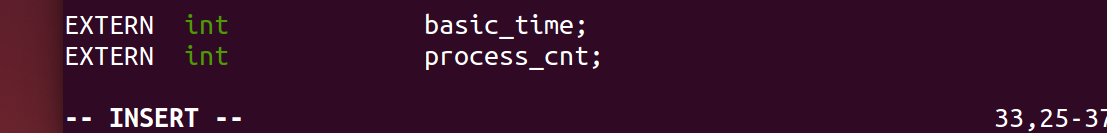


可以看到成功达到了我们的预期效果；

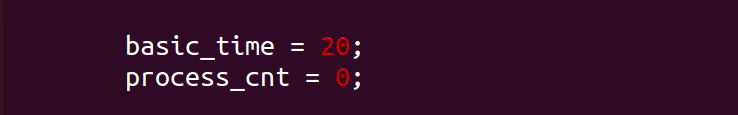
但是，最后BC进程出现了饥饿的现象，这是我们所不希望的，所以下面我们通过改进代码来解决这个问题；

首先解释以下为什么会出现饥饿，这是由于我们每次在遍历p\_proc\_table数组的时候，都是从头开始，如果在最低级别的队列中，则排号在前面的进程将永远抢先于排号在后面的进程，所以我们采取的解决方案是，添加一个全局变量process\_cnt，用它来记录当前遍历的位置，下次接着从这个位置开始，而实现循环迭代则只需要使用一个模加操作即可完成；

首先在/include/global.h中定义这个变量：



然后在kernel/main.c中初始化：



重点是要修改时钟中断处理程序，由于前两级队列都会沉淀到更低优先级队列，所以不会产生饥饿现象，我们可以暂时不做修改，故仅对最低级队列进行修改；

PUBLIC **void** clock\_handler(**int** irq)

{

    ticks++;

    p\_proc\_ready->ticks--;

    PROCESS\* p;

**int** have\_high;

    if (k\_reenter != 0) {

        return;

    }

    if(p\_proc\_ready->priority == 0){

*/\*如果当前进程在最高优先级队列且时间片未用完，则一定不会被抢占\*/*

            if(p\_proc\_ready->ticks != 0) return ;

*/\*时间片用完\*/*

        p\_proc\_ready->ticks = basic\_time \* 2;

        p\_proc\_ready->priority = 1;

        schedule();

    }

    else if(p\_proc\_ready->priority == 1){

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

            p\_proc\_ready->ticks = basic\_time \* 3;

            p\_proc\_ready->priority = 2;

            schedule();

            return ;

        }

        have\_high = 0;

*/\*检查是否有优先级更高的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(p->priority == 0){

                have\_high = 1;

                break;

            }

        }

        if(have\_high){

*/\*如果有优先级更高的进程\*/*

*/\*则发生抢占\*/*

            p\_proc\_ready = p;

        }

        }

    else if(p\_proc\_ready->priority == 2){

**int** have0 = 0;

**int** have1 = 0;

        if(p\_proc\_ready->ticks == 0){

*/\*如果时间片用完，则直接调度\*/*

*/\*由于进程在最低优先级队列，不会降级，仅恢复时间片\*/*

                        p\_proc\_ready->ticks = basic\_time \* 3;

                        schedule();

            if(p\_proc\_ready->priority == 2){

*/\*如果新进程的优先级还是2，大概率会出问题\*/*

                while(1){

                    process\_cnt = (process\_cnt + 1) % NR\_TASKS;

                    p\_proc\_ready = proc\_table + process\_cnt;

                    if(p\_proc\_ready->priority == 2) break;

                }

            }

                        return ;

                }

*/\*检查是否有优先级为0的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

                        if(p->priority == 0){

                                have0 = 1;

                                break;

                        }

                }

*/\*检查是否有优先级为1的进程\*/*

        for(p = proc\_table; p < proc\_table+NR\_TASKS; p++){

            if(have0) break;

            if(p->priority == 1){

                have1 = 1;

                break;

            }

        }

        if(have0 || have1){

*/\*如果有更高优先级的进程就发生抢占\*/*

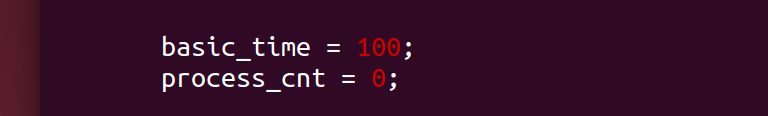
            p\_proc\_ready = p;

        }

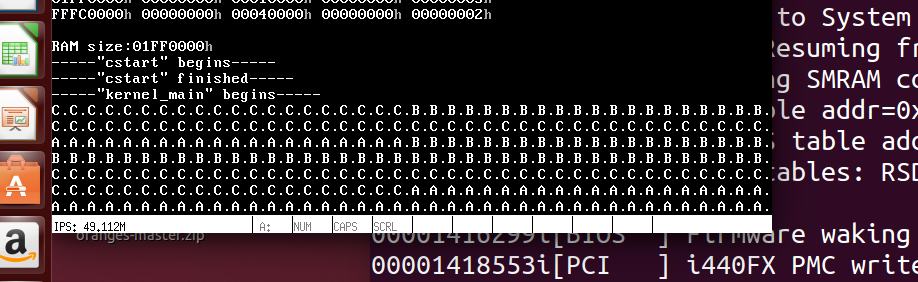
        }

}

此外就是basic\_time如果设置的过小，则会严重影响实验效果，所以我们做如下调整：



运行一下查看效果：



可以看到开始的时候仍然和之前一样，但是后边则是ABC交替出现，并没有出现某个进程饥饿的情况。