进程控制实验

班级 122030704　学号 12208990406　姓名 刘宇轩

目标	目标 3 得分	目标 4 得分
自评分	95	95
批改分		

一、实验目的

1、掌握 Linux 下的多进程编程技术，并能够按照实验要求进行软硬件的实现及验证。

2、能够通过自主学习学习实验相关知识，并解决实验中遇到的具体问题。

二、实验结果及分析

1. 执行 process.c

```
shiyanlou@6757c900680755efb2c5c338:~/oslab$ ./process
Child process 1 is running.
Child process 2 is running.
Child process 3 is running.
This process's pid is 663
Pid of child process 1 is 664
Pid of child process 2 is 665
Pid of child process 3 is 666
Pid of child process 4 is 667
Child process 4 is running.
Child process 2 is end.
Child process 4 is end.
Child process 3 is end.
Child process 1 is end.
end.
shiyanlou@6757c900680755efb2c5c338:~/oslab$
```

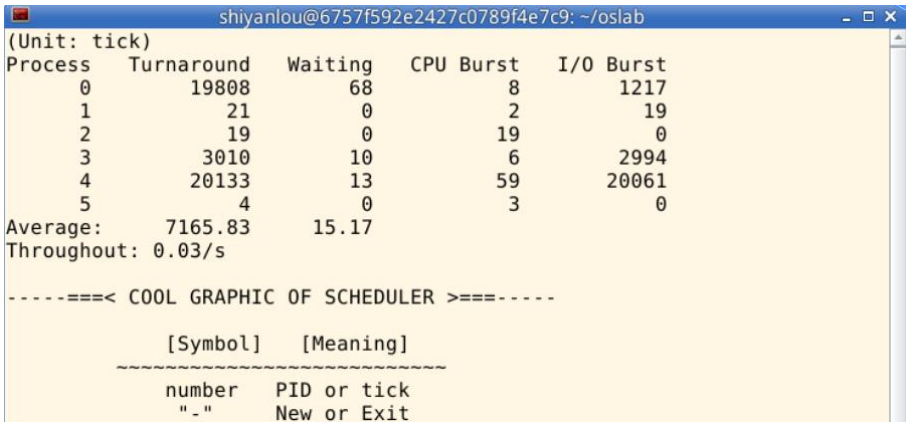
成功运行。

2. 建立日志

```
12  R  5397
13  N  5398
13  J  5398
14  N  5399
14  J  5399
15  N  5399
15  J  5400
16  N  5400
16  J  5401
12  W  5401
16  R  5402
16  J  5407
15  R  5407
15  J  5412
14  R  5412
14  W  5412
13  R  5412
13  J  5417
16  R  5417
16  J  5422
```

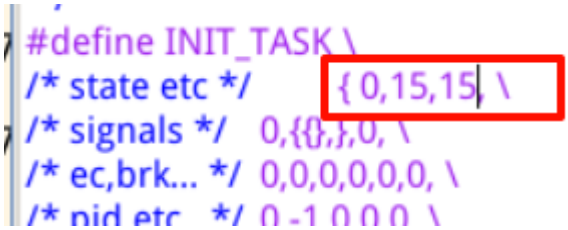
成功在相关的进程状态变更点记录日志。

3. 成功用 stat_log.py 进行日志分析



成功运行输出。

4. 修改时间片并且运行输出。

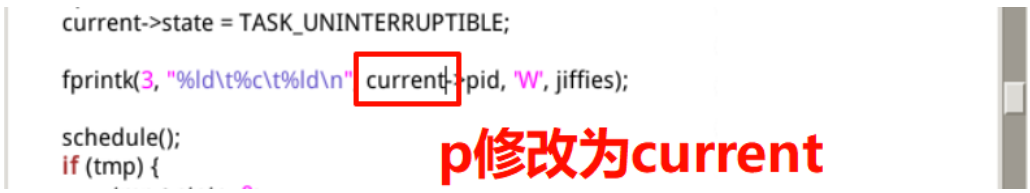


三、问题及解决方法

1. 出现 make all 编译不通过

```
sched.c: In function `schedule':
sched.c:120: warning: implicit declaration of function `fprintk'
sched.c:146: error: syntax error before ';' token
sched.c: In function `sys_pause':
sched.c:157: error: syntax error before ';' token
sched.c: In function `sleep_on':
sched.c:177: error: request for member `pid' in something not a structur
e or union
make[1]: *** [sched.o] 错误 1
make[1]:正在离开目录 `/home/shiyanlou/oslab/linux-0.11/kernel'
make: *** [kernel/kernel.o] 错误 2
shiyanlou@6757f592e2427c0789f4e7c9:~/oslab/linux-0.11$ gedit kernel/prin
```

修改 kernel/sched.c 中 sleep_on 函数 fprintk 函数内容



```
    if (current->pid != task[next]->pid) {
        if (current->state == TASK_RUNNING){
            fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);
        }
        fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);
    }

    switch_to(next);
}
```

将()改为{}

2. process.log 日志有错误，错误输出如下：

process.log

```
1   N   48
1   J   48
0   J   48
1   R   48
2   N   49
2   J   49
1   W   49
2   R   49
3   N   64
3   J   64
2   J   64
3   R   64
```

并且在运行 stat_log.py 后出现错误如下：

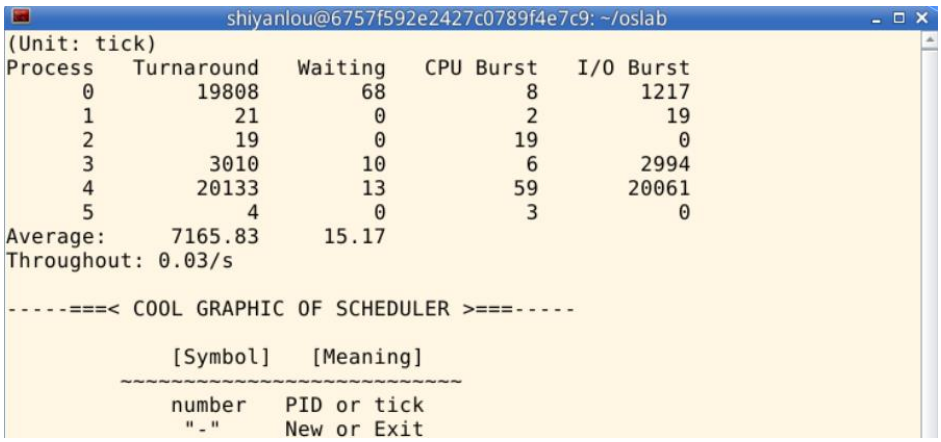
```
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      0             8          0            8            0
      1             1          0            1            0
      2            15          0           15            0
      3             1          0            0            0
Average:         6.25       0.00
Throughout: 16.67/s

-----===< COOL GRAPHIC OF SCHEDULER >===-----

            [Symbol]    [Meaning]
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              number    PID or tick
               "-"      New or Exit
               "#"        Running
               "|"         Ready
```

修改：

```
    __asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");
    ltr(0);
    lldt(0);
    outb_p(0x36,0x43);          /* binary, mode 3, LSB/MSB, ch 0 */
    outb_p(LATCH & 0xff , 0x40);    /* LSB */
    outb_p(LATCH >> 8 , 0x40);  /* MSB */
    set_intr_gate(0x20,&timer_interrupt);
    outb(inb_p(0x21)&~0x01,0x21);
    set_system_gate(0x80,&system_call);
}
```

将outb修改为outb_p

之后重新编写内核，尝试每次在关闭 bochs 前，执行 sync，刷新 cache，确保文件确实写入了磁盘，之后运行成功。



四、实验思考

1. 优化进程控制的调度算法

进程调度算法是操作系统中核心的一部分，其优化有助于提高系统的整体效率。在实验中，常见的调度算法如先来先服务（FCFS）、最短作业优先（SJF）和时间片轮转（RR）等都有不同的优缺点。比如，FCFS 算法实现简单，但可能导致“等待时间过长”问题，而 RR 算法则可以较好地解决这个问题，但可能导致进程切换的开销较大。所以优化调度算法的关键在于根据系统需求选择合适的策略。

2. 进程控制的难点

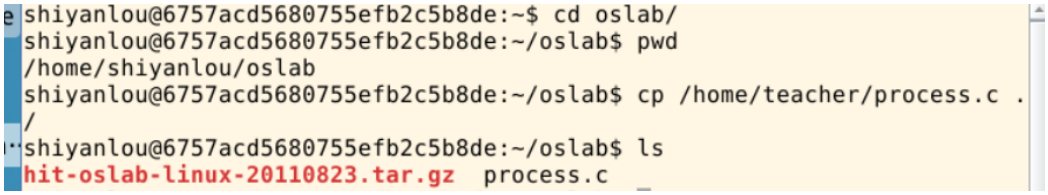
在实际操作系统中，进程控制面临许多复杂的问题，例如死锁、进程同步与互斥、资源分配与管理等。死锁是一个比较典型且难以避免的问题，需要通过资源分配图、银行家算法等手段来预防或解决。进程同步与互斥是为了保证多进程环境下数据的一致性和程序的正确执行，常用的技术包括信号量、互斥锁等。如何合理设计和实现这些机制，避免系统资源浪费和数据不一致，是进程控制中的一个关键挑战。

3. 提高进程的执行效

提高进程执行效率可以通过多种手段。1.通过合理的进程调度策略，确保系统资源的最大化利用。2.避免不必要的进程切换，减少上下文切换的开销。3.适当使用缓存技术、减少进程间的阻塞等待，增强进程并发执行的效率。4.可以通过优化系统调用，减少进程在等待资源时的延迟，提高系统响应速度。

五、实验步骤

1. 将 process.c 复制过来



2. 编写 process.c 文件

shiyanlou@6757acd5680755efb2c5b8de: ~/oslab

```
#define __LIBRARY__
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <sys/times.h>
#include <sys/wait.h>

#define HZ      100

void cpuio_bound(int last, int cpu_time, int io_time);

int main(int argc, char *argv[])
{
    pid_t pid1, pid2, pid3, pid4;

    /* 第1个子进程 */
    /* 负数->fork失败;父进程返回的是子进程的pid(>0)&&子进程返回0->fork
成功*/
    pid1 = fork();
    if (pid1 < 0) /* 负数->错误。 */
        printf("error in fork!,  errno=%d\n", pid1);
    else if (pid1 == 0) /* 0->子进程。 */
    {
        printf("Child process 1 is running.\n");
        cpuio_bound(10, 1, 0); /* 占用10秒的CPU时间 */
        printf("Child process 1 is end.\n");
        return 0;
    }

"process.c" 123L, 3279C                                   1,1           顶端
```

shiyanlou@6757acd5680755efb2c5b8de: ~/oslab

```
    /* 第2个子进程 */
    pid2 = fork();
    if (pid2 < 0)
        printf("error in fork!,  errno=%d\n", pid2);
    else if (pid2 == 0)
    {
        printf("Child process 2 is running.\n");
        cpuio_bound(10, 0, 1); /* 以IO为主要任务 */
        printf("Child process 2 is end.\n");
        return 0;
    }

    /* 第3个子进程 */
    pid3 = fork();
    if (pid3 < 0)
        printf("error in fork!,  errno=%d\n", pid3);
    else if (pid3 == 0)
    {
        printf("Child process 3 is running.\n");
        cpuio_bound(10, 1, 1); /* CPU和IO各1秒钟轮回 */
        printf("Child process 3 is end.\n");
        return 0;
    }

    /* 第4个子进程 */
    pid4 = fork();
    if (pid4 < 0)
        printf("error in fork!,  errno=%d\n", pid4);
                                                  56,1          28%
```

```
shiyanlou@6757acd5680755efb2c5b8de: ~/oslab
    else if (pid4 == 0)
    {
        printf("Child process 4 is running.\n");
        cpuio_bound(10, 1, 9); /* IO时间是CPU的9倍 */
        printf("Child process 4 is end.\n");
        return 0;
    }

    printf("This process's pid is %d\n", getpid());
    printf("Pid of child process 1 is %d\n", pid1);
    printf("Pid of child process 2 is %d\n", pid2);
    printf("Pid of child process 3 is %d\n", pid3);
    printf("Pid of child process 4 is %d\n", pid4);

    wait(NULL);
    wait(NULL);
    wait(NULL);
    wait(NULL);

    printf("end.\n");
    return 0;
}
```

编写并执行 process.c，没用问题

```
shiyanlou@6757c900680755efb2c5c338:~/oslab$ ./process
Child process 1 is running.
Child process 2 is running.
Child process 3 is running.
This process's pid is 663
Pid of child process 1 is 664
Pid of child process 2 is 665
Pid of child process 3 is 666
Pid of child process 4 is 667
Child process 4 is running.
Child process 2 is end.
Child process 4 is end.
Child process 3 is end.
Child process 1 is end.
end.
shiyanlou@6757c900680755efb2c5c338:~/oslab$
```

3. 在 init/main.c 文件中添加代码，

```
    move_to_user_mode();

    setup((void *) &drive_info);
    (void) open("/dev/tty0",O_RDWR,0);
    (void) dup(0);
    (void) dup(0);
    (void) open("/var/process.log",O_CREAT|O_TRUNC|O_WRONLY,0666);

    if (!fork()) {          /* we count on this going ok */
```

4. 在 kernel/printk.c 中添加 fprintk 函数代码

```
#include "linux/sched.h"
#include "sys/stat.h"
#include <stdarg.h>
#include <stddef.h>

#include <linux/kernel.h>

static char buf[1024];
static char logbuf[1024];
```

```
int fprintk(int fd, const char *fmt, ...)
{
    va_list args;
    int count;
    struct file * file;
    struct m_inode * inode;

    va_start(args, fmt);
    count=vsprintf(logbuf, fmt, args);
    va_end(args);
    if (fd < 3)
    {
        __asm__("push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $logbuf\n\t"
            "pushl %1\n\t"
            "call sys_write\n\t"
            "addl $8,%%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            ::"r" (count),"r" (fd):"ax","cx","dx");
    }
    else
    {
        if (!(file=task[0]->filp[fd]))
            return 0;
        inode=file->f_inode;

        __asm__("push %%fs\n\t"
            "push %%ds\n\t"
            "pop %%fs\n\t"
            "pushl %0\n\t"
            "pushl $logbuf\n\t"
            "pushl %1\n\t"
            "pushl %2\n\t"
            "call file_write\n\t"
            "addl $12,%%esp\n\t"
            "popl %0\n\t"
            "pop %%fs"
            ::"r" (count),"r" (file),"r" (inode):"ax","cx","dx");
    }
    return count;
}
```

5. 编写 kernel/sched.c 文件

Sched.c 函数

```
        (*p)->state==TASK_INTERRUPTIBLE) {
            (*p)->state=TASK_RUNNING;
            fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
        }
    }
```

```
        }

        if (current->pid != task[next]->pid) {
                if (current->state == TASK_RUNNING) (
                        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies);
                )
                fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);
        }

        switch_to(next);
}
```

Sleep_on 函数

```
        current->state = TASK_UNINTERRUPTIBLE;

        fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'W', jiffies);

        schedule();
        if (tmp) {
                tmp->state=0;
                fprintk(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
        }
}

void interruptible_sleep_on(struct task_struct **p)
```

interruptible_sleep_on 函数

```
repeat:   current->state = TASK_INTERRUPTIBLE;

        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);

        schedule();
        if (*p && *p != current) {
                (**p).state=0;
                fprintk(3, "%ld\t%c\t%ld\n", (**p).pid, 'J', jiffies);
                goto repeat;
        }
        *p=NULL;
        if (tmp) {
                tmp->state=0;
                fprintk(3, "%ld\t%c\t%ld\n", tmp->pid, 'J', jiffies);
        }
```

sys_pause 函数

```
int sys_pause(void)
{
        if (current->state != TASK_INTERRUPTIBLE) (
                fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
        )

        current->state = TASK_INTERRUPTIBLE;
        schedule();
        return 0;
}
```

wake_up 函数

```
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        (**p).state=0;
        fprintk(3, "%ld\t%c\t%ld\n", (*p)->pid, 'J', jiffies);
        *p=NULL;
    }
}
```

6. kernel/exit.c 中 sys_waitpid 函数

```
        return 0;
        current->state=TASK_INTERRUPTIBLE;
        fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies);
        schedule();
        if (!(current->signal &= ~(1<<(SIGCHLD-1))))
            goto repeat;
```

do_exit 函数

```
        kill_session();
    current->state = TASK_ZOMBIE;
    fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies);
    current->exit_code = code;
    tell_father(current->father);
    schedule();
```

7. 新建进程，编写 kernel/fork.c 的 copy_process 函数

```
    p->cutime = p->cstime = 0;
    p->start_time = jiffies;
    fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies);
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    p->state = TASK_RUNNING;  /* do this last, just in case */
    fprintk(3, "%ld\t%c\t%ld\n", p->pid, 'J', jiffies);
    return last_pid;
}
```

8. 编译内核成功

```
Setup is 312 bytes.
System is 123748 bytes.
rm system.tmp
rm tools/kernel -f
sync
shiyanlou@6757f592e2427c0789f4e7c9:~/oslab/linux-0.11$
```

9. 编译并运行 process.c

```
[/usr/root]# gcc -o process process.c
[/usr/root]# ./process
This process's pid is 12
Pid of child process 1 is 13
Pid of child process 2 is 14
Pid of child process 3 is 15
Pid of child process 4 is 16
Child process 4 is running.
Child process 3 is running.
Child process 2 is running.
Child process 1 is running.
Child process 2 is end.
Child process 4 is end.
Child process 3 is end.
Child process 1 is end.
end.
[/usr/root]# sync
```

内核会记录日志到 /var/process.log 文件中

将 stat_log.py 复制到 oslab 目录中，并且增加可执行权限，挂在后分析日志

```
shiyanlou@6757f592e2427c0789f4e7c9:~/oslab$ cp ./linux-0.11/stat_log.py
./
shiyanlou@6757f592e2427c0789f4e7c9:~/oslab$ chmod +x stat_log.py
shiyanlou@6757f592e2427c0789f4e7c9:~/oslab$ sudo ./mount-hdc
shiyanlou@6757f592e2427c0789f4e7c9:~/oslab$ ./stat_log.py hdc/var/proces
s.log 0 1 2 3 4 5 -g | less
```

日志如下：

```
shiyanlou@6757f592e2427c0789f4e7c9: ~/oslab
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      0         19808         68            8         1217
      1            21          0            2           19
      2            19          0           19            0
      3          3010         10            6         2994
      4         20133         13           59        20061
      5             4          0            3            0
Average:     7165.83      15.17
Throughout: 0.03/s

-----===< COOL GRAPHIC OF SCHEDULER >===-----

             [Symbol]   [Meaning]
          ~~~~~~~~~~~~~~~~~~~~~~~~~~
             number   PID or tick
              "-"     New or Exit
```

10. 修改时间片，只要修改进程 0 初始化时的时间片（counter）和优先级（priority）即可

执行

```
shiyanlou@6757f592e2427c0789f4e7c9:~/oslab$ gedit linux-0.11/include/lin
ux/sched.h
```

找到下面内容，其中圈出来的三个数字分别对应 state、counter 和 priority，

我们只需要修改第三个值即可，就是修改时间片的初始值即可。

```
#define INIT_TASK \
/* state etc */      { 0,15,15, \
/* signals */   0,{{},},0, \
/* ec,brk... */ 0,0,0,0,0,0, \
/* pid etc.. */ 0,-1,0,0,0, \
```

下面是修改时间片初始值为 5，20，50，100，500 后的结果

时间片：5

```
shiyanlou@6757f592e2427c0789f4e7c9: ~/oslab
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      6           303          1           14          288
      7            61          1           60            0
      8           115          1          114            0
      9            35          0           34            0
     10            81          1           80            0
Average:       119.00       0.80
Throughout: 1.65/s
```

时间片：20

```
shiyanlou@6757f592e2427c0789f4e7c9: ~/oslab
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      6           303          1           11          290
      7            61          1           60            0
      8           115          0          115            0
      9            35          0           34            0
     10            81          0           81            0
Average:       119.00       0.40
Throughout: 1.65/s
```

时间片：50

```
shiyanlou@6757f592e2427c0789f4e7c9: ~/oslab
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      6           303          0           12          290
      7            61          0           61            0
      8           115          0          115            0
      9            34          0           33            0
     10            82          0           81            0
Average:       119.00       0.00
Throughout: 1.65/s
```

时间片：100

```
shiyanlou@6757f592e2427c0789f4e7c9: ~/oslab
(Unit: tick)
Process    Turnaround    Waiting    CPU Burst    I/O Burst
      6           303         14           14          275
      7            61          0           60            0
      8           115          1          114            0
      9            35          0           34            0
     10            81          1           80            0
Average:       119.00       3.20
Throughout: 1.65/s
```

时间片：500

```
(Unit: tick)
Process   Turnaround   Waiting   CPU Burst   I/O Burst
      6          303        13          14         275
      7           61         0          60           0
      8          115         0         114           0
      9           34         1          33           0
     10           82         0          81           0
Average:      119.00      2.80
Throughout: 1.65/s
```

六、实验收获

1. 通过实验，进一步了解了操作系统中进程的创建、调度、终止等过程。掌握了如何通过系统调用（如 fork()、exec()等）来实现进程管理。
2. 通过实现和测试不同的进程调度算法，理解了各类调度算法的原理及优缺点，例如先来先服务（FCFS）、最短作业优先（SJF）、轮转调度（RR）等。
3. 在实验过程中，了解了操作系统如何通过进程表、调度队列等数据结构来管理进程，并学会了如何通过编程模拟这些机制。
4. 在实验中，学习了如何实现资源分配策略，以及如何避免资源竞争、死锁等问题。
5. 通过实际操作，深入理解了系统调用如何在内核与用户空间之间进行交互，掌握了进程控制相关的常用系统调用的实现与应用。
6. 通过实现和调试进程调度算法，锻炼了自己的 C/C++编程能力，同时提升了问题分析和系统设计能力。

七、请了解熟悉以余祖胜烈士为代表的校友校史校情，结合我们的课程或专业，谈谈如何在我们目前的学习生活中发扬传承重庆理工大学“抗战文化、红岩精神、兵工基因”的校本文化？

我们学校有着以余祖胜烈士为代表的光荣校友，他们所展现出的抗战文化、红岩精神和兵工基因，是我们宝贵的精神财富。在计算机操作系统课程和计算机科学与技术专业学习中，我要从多个方面传承发扬校本文化。在理论知识学习上，以坚韧不拔的抗战文化精神面对复杂抽象的知识，像红岩英烈一样坚定信念去攻克难题；在实验实践环节，结合红岩精神勇于创新实践，以兵工基因的责任感确保操作安全规范。在构建知识体系时，用抗战文化的团结协作精神整合不同课程知识，以红岩精神的崇高思想境界追求先进技术。参与科研与竞赛活动时，发扬兵工基因的钻研精神，以红岩精神的不屈意志应对困难。日常学习生活中，我会积极参与校园文化活动和社团组织活动，分享余祖胜烈士事迹和校本文化内涵，营造良好的学习交流氛围，通过编程等方式弘扬校本文化，让校本文化深深影响自己和身边的每一位同学。

八、参考资料

1. https://blog.csdn.net/chenjiebin/article/details/144107597
2. https://blog.csdn.net/qq_42518941/article/details/119061014
3. https://blog.csdn.net/qq_33767966/article/details/143611555
4. 赵炯博士的《Linux 内核完全注释》