

基于LoongArch架构的xv6实验自动评判系统设计报告

2023全国大学生计算机系统能力大赛操作系统设计赛-功能挑战赛

0x3f

黄万丰 黄灿彬

一、设计简介

对现有的龙芯xv6实验，设计一个自动批改系统，学生提交指定实验的代码后可以获得是否通过的判定结果。这次实验之后或可用于高校操作系统课的作业评测。实验内容具体见本项目根目录下的龙芯-xv6操作系统实验-22-11-13(整洁-无修订).pdf。

二、比赛题目分析和相关资料调研

为了能对学生提交的实验代码自动评判，我们需要能对qemu仿真下的xv6系统进行观察和调试。主要分成两个自动化环节，一个负责与xv6的shell输入输出交互，另一个则实现远程gdb自动调试。其中前者可以在应用层模拟用户的操作，后者可以观察代码的运行细节。

实现本系统时，参考研究了使用xv6作为教学操作系统的高校的课程，详见<https://pdos.csail.mit.edu/6.828/2022/>。

三、系统框架设计

该评测系统的整体设计包括：**目标实验系统设计** 以及 **测试框架**组成。

3.1 目标实验系统设计

本次比赛，挑选龙芯xv6操作系统实验指导书——龙芯-xv6操作系统实验-22-11-13(整洁-无修订).pdf 中的四个小实验作为样例展示。

采用的是使用 `git` 代码协同管理的方式，设想在原始实验环境基础上，创建四个分支分别为Ex1、Ex2、Ex3、Ex4。

在此基础上在每个分支仓库中添加**对应实验的测试脚本文件**，存放在Test文件夹下，这样就准备好了操作系统实验最初的环境，将相应的仓库推送到远程仓库中存放，学生学习对应实验的时候只需要克隆远程仓库对应分支代码进行修改即可。

以下是克隆远程仓库Ex1分支文件后，在本地使用 `git branch -a` 查看所有分支，如下：

```
● kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git branch -a
* Ex1
remotes/origin/Ex1
remotes/origin/Ex2
remotes/origin/Ex3
remotes/origin/Ex4
remotes/origin/HEAD -> origin/Ex1
```

3.2 测试框架

分为 学生本地代码测试框架 以及 远程提交服务器测试框架。

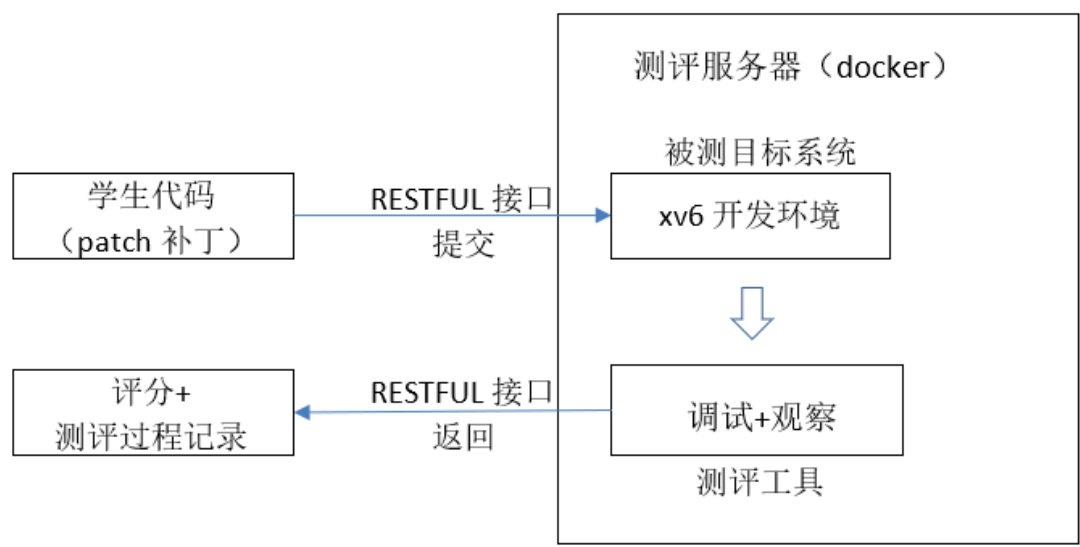
学生本地代码测试框架

学生克隆实验准备好的远程仓库分支进行实验，实验完成后，进入Test目录中执行相应的脚本文件，即可从shell终端中获取实验评测结果。

远程提交服务器测试框架

测试框架以docker形式提供对外服务，用户通过网络向服务器提交xv6实验代码——patch补丁包，在对应的实验编号的前端UI界面上提交patch；测试系统自动完成测试通过网络返回测评结果及测评过程记录。

系统框架图如下：



注：本地代码测试框架截至初赛提交日期已经完成，而远程服务器测试框架只是做了初步的设想验证，没有完全实现自动化评测系统。

四、实现描述

项目文件结构如下：

```
1 .
2 |— imgs
3 |— kernel
4 |— Makefile
5 |— mkfs
6 |— qemu-loongarch-runenv
7 |— README
8 |— README.md
9 |— Test
10 |— test.sh
11 |— user
```

Test文件夹即为存放测试脚本的位置。

4.1 调试与观察手段

4.1.1 与xv6终端的交互

使用qemu的-serial参数将xv6串口重定向到tcp端口，因此可以通过脚本或程序与xv6的终端进行互操作，从而为实现自动评判提供辅助。

服务器端：启动qemu

在qemu启动时使用-serial tcp::XXXX,server就可以将标准输入输出串口设备转入到TCP服务器端口XXXX上。例如下面的命令，将会在TCP:4556端口实现输入输出重定向。

```
1 ../../qemu-loongarch-runenv/qemu/x86_64/qemu-system-loongarch64\
2 -m 1G -smp 1 -bios \
3 ../../qemu-loongarch-runenv/loongarch_bios_0310_debug.bin\
4 -kernel ../../kernel/kernel \
5 -initrd ../../qemu-loongarch-runenv/busybox-rootfs.img \
6 -append 'root=/dev/ram console=ttyS0,115200 rdinit=/init' \
7 -vga none -nographic \
8 -L ../../qemu-loongarch-runenv\
9 -serial tcp::4556,server,nowait
```

运行效果如下：

```
1 rex@rex-virtual-machine:~/test/T1$ ~/xv6/xv6-loongarch-exp/qemu-loongarch-
runenv/qemu/x86_64/qemu-system-loongarch64 -m 1G -smp 1 -bios ~/xv6/xv6-
loongarch-exp/qemu-loongarch-runenv/loongarch_bios_0310_debug.bin -kernel
~/xv6/xv6-loongarch-exp/kernel/kernel -initrd ~/xv6/xv6-loongarch-exp/qemu-
loongarch-runenv/busybox-rootfs.img -append 'root=/dev/ram
console=ttyS0,115200 rdinit=/init' -vga none -nographic -L ~/xv6/xv6-
loongarch-exp/qemu-loongarch-runenv -serial tcp::4556,server,nowait
2 QEMU 6.2.50 monitor - type 'help' for more information
3 (qemu)
```

客户端：建立TCP连接

使用如下代码，可以连接本地TCP的4556端口：

```
1 rex@rex-virtual-machine:~/test/T1$ nc 127.0.0.1 4556
```

运行效果如下：

```
1 rex@rex-virtual-machine:~/test/T1$ nc 127.0.0.1 4556
2 ls
3 ls
4 .          1 1 1024
5 ..         1 1 1024
6 README    2 2 2226
7 cat       2 3 20576
8 echo      2 4 19576
9 forktest  2 5 11776
10 grep      2 6 23104
11 init      2 7 20296
12 kill      2 8 19464
13 ln        2 9 19360
14 ls        2 10 22424
15 mkdir     2 11 19552
16 rm        2 12 19536
17 sh        2 13 35208
18 stressfs  2 14 20408
19 usertests 2 15 125400
20 my-app     2 16 19080
21 print-pid 2 17 19144
22 pcputid    2 18 19144
23 sh_rw_nolock 2 19 19584
24 sh_rw_lock 2 20 19696
25 loop      2 21 19728
26 prio-sched 2 22 20272
27 console   3 23 0
28 $
```

4.1.2 与qemu控制台交互

除了需要和xv6的终端进行交互外，我们可能还要控制qemu的仿真过程，因此也需要与之进行交互。

服务器端：启动qemu

交互使用qemu -monitor参数，将控制台重定向到tcp端口。可以使用以下命令，将Qemu的标准输入输出转向到TCP端口4557上：

```
1 ~/xv6/xv6-loongarch-exp/qemu-loongarch-runenv/qemu/x86_64/qemu-system-
  loongarch64 -m 1G -smp 1 -bios ~/xv6/xv6-loongarch-exp/qemu-loongarch-
  runenv/loongarch_bios_0310_debug.bin -kernel ~/xv6/xv6-loongarch-
  exp/kernel/kernel -initrd ~/xv6/xv6-loongarch-exp/qemu-loongarch-
  runenv/busybox-rootfs.img -append 'root=/dev/ram console=ttyS0,115200
  rdinit=/init' -vga none -nographic -L ~/xv6/xv6-loongarch-exp/qemu-loongarch-
  runenv -monitor tcp::4557,server
```

运行效果如下：



```

1 rex@rex-virtual-machine:~/test/T1$ ~/xv6/xv6-loongarch-exp/qemu-loongarch-
runenv/qemu/x86_64/qemu-system-loongarch64 -m 1G -smp 1 -bios ~/xv6/xv6-
loongarch-exp/qemu-loongarch-runenv/loongarch_bios_0310_debug.bin -kernel
~/xv6/xv6-loongarch-exp/kernel/kernel -initrd ~/xv6/xv6-loongarch-exp/qemu-
loongarch-runenv/busybox-rootfs.img -append 'root=/dev/ram
console=ttyS0,115200 rdinit=/init' -vga none -nographic -L ~/xv6/xv6-
loongarch-exp/qemu-loongarch-runenv -monitor tcp::4557,server
2 .....
3 entry kernel ...
4 cpu0: starting xv6
5 init: starting sh
6 $ ls
7 .                1 1 1024
8 ..               1 1 1024
9 README          2 2 2226
10 cat             2 3 20576
11 echo           2 4 19576
12 forktest       2 5 11776
13 grep           2 6 23104
14 init           2 7 20296
15 kill           2 8 19464
16 ln             2 9 19360
17 ls             2 10 22424
18 mkdir          2 11 19552
19 rm             2 12 19536
20 sh             2 13 35208
21 stressfs       2 14 20408
22 usertests      2 15 125400
23 my-app         2 16 19080
24 print-pid      2 17 19144
25 pcpid          2 18 19144
26 sh_rw_nolock   2 19 19584
27 sh_rw_lock     2 20 19696
28 loop          2 21 19728
29 prio-sched     2 22 20272
30 console       3 23 0
31 $

```

客户端：连接qemu服务器

客户端通过nc工具可以连接到qemu的服务器上，从而为程序控制qemu的行为提供了可能。

以下命令可以让客户端连接到qemu服务器：

```
1 nc 127.0.0.1 4557
```

运行效果如下：

```

1 rex@rex-virtual-machine:~/test/T1$ nc 127.0.0.1 4557
2 QEMU 6.2.50 monitor - type 'help' for more information
3 (qemu) info registers
4 info registers
5 PC=900000000201064 FCSR0 0x00000000 fp_status 0x00
6 GPR00: r0 0000000000000000 r1 9000000002010c0 r2 0000000000000000 r3
900000000030a310

```

```

7  GPR04: r4 90000000030d010 r5 90000000030d018 r6 0000000000000001 r7
0000000000000005
8  GPR08: r8 0505050505050505 r9 0505050505050505 r10 0505050505050505 r11
0000000000000005
9  GPR12: r12 0000000000000000 r13 0000000000000000 r14 0000000000000001 r15
90000000030d010
10 GPR16: r16 0505050505050505 r17 0505050505050505 r18 0505050505050505 r19
0505050505050505
11 GPR20: r20 0505050505050505 r21 0505050505050505 r22 90000000030a350 r23
0000000000000003
12 GPR24: r24 90000000030a830 r25 90000000020c1a8 r26 90000000030d010 r27
90000000030d018
13 GPR28: r28 0000000000000013 r29 00000000bfe8f7f8 r30 0000000000000000 r31
00000000bfe8f7d0
14 CRMD=00000000000000b4
15 PRMD=0000000000000004
16 EUEN=0000000000000000
17 ESTAT=0000000000000000
18 ERA=900000000201064
19 BADV=0000000000000000
20 BADI=0000000029c06061
21 EENTRY=900000000208000
22 PRCFG1=0000000000000000, PRCFG2=0000000000000000, PRCFG3=0000000000000000
23 TLBRETRY=90000000020a000
24 TLBRBADV=0000000000fbff8
25 TLBRERA=900000000203a34
26 f0 0000000000000000 f1 0000000000000000 f2 0000000000000000 f3
0000000000000000
27 f4 0000000000000000 f5 0000000000000000 f6 0000000000000000 f7
0000000000000000
28 f8 0000000000000000 f9 0000000000000000 f10 0000000000000000 f11
0000000000000000
29 f12 0000000000000000 f13 0000000000000000 f14 0000000000000000 f15
0000000000000000
30 f16 0000000000000000 f17 0000000000000000 f18 0000000000000000 f19
0000000000000000
31 f20 0000000000000000 f21 0000000000000000 f22 0000000000000000 f23
0000000000000000
32 f24 0000000000000000 f25 0000000000000000 f26 0000000000000000 f27
0000000000000000
33 f28 0000000000000000 f29 0000000000000000 f30 0000000000000000 f31
0000000000000000
34 (qemu)

```

4.1.3 gdb自动调试手段

如果需要更详细的程序运行控制，则需要借助gdb的帮助——设置断点和条件然后观察中间状态，从而判定程序行为是否如预设的那样。这涉及两个内容，一个是gdb调试命令如何自动执行；另一个是如何启动qemu中的gdb服务器，以及如何用gdb客户端远程连接到gdb服务器。

gdb测试脚本

gdb允许编写脚本来完成调试任务，而无需手动地逐条执行这些命令。例如我们编写如下所示的mycmd.gdb脚本，它将远程链接到调试服务器，然后用file命令指出调试影像是kernel文件，然后用b命令（break）将断点设置在main()入口处，接着用c命令（continue）继续运行（将运行到main()入口处停下），接着用l命令查看当前代码，并用p命令（print）打印main函数地址，最后退出调试。

```
1 target remote :1234
2 file ../kernel/kernel
3 b main
4 c
5 l
6 p main
7 quit
```

服务器端：启动qemu的gdb调试

由于qemu支持gdb调试功能，并提供有命令行选项用于开启gdb服务器。下面的命令中-gdb tcp::1234 -S就是用于启动gdb server服务端的。

```
1 ./qemu/x86_64/qemu-system-loongarch64 -m 1G -smp 1 -bios
  ./loongarch_bios_0310_debug.bin -kernel ../kernel/kernel -initrd busybox-
  rootfs.img -append 'root=/dev/ram console=ttyS0,115200 rdinit=/init' -vga none
  -nographic -gdb tcp::1234 -S
```

客户端：连接到gdb服务器

这里所用的gdb客户端必须能理解LA64架构，因此不能使用宿主机上的x86-64版本的gdb程序。必须使用龙芯LA64版本的gdb客户端来连接到gdb服务端。下面我们自动执行前面编写好的脚本mycmd.gdb，并通过“>”输出重定向将结果保存到mygdb.output文件，实现了与gdb的交互。

```
1 loongarch64-unknown-linux-gnu-gdb --batch --command mycmd.gdb > mycmd.output
```

运行效果如下：

```
1 rex@rex-virtual-machine:~/xv6/xv6-loongarch-exp/qemu-loongarch-runenv$
  loongarch64-unknown-linux-gnu-gdb --batch --command mycmd.gdb > mycmd.output
2 rex@rex-virtual-machine:~/xv6/xv6-loongarch-exp/qemu-loongarch-runenv$ cat
  mygdb.output
3 0x000000001c000000 in ?? ()
4 Breakpoint 1 at 0x900000000200070: file kernel/main.c, line 16.
5
6 Breakpoint 1, main () at kernel/main.c:16
7 16     if(cpuid() == 0){
8 11
9 12 // entry.S jumps here on stack0.
10 13 void
11 14 main()
12 15 {
13 16     if(cpuid() == 0){
14 17         printf("cpu0: starting xv6\n");
15 18         consoleinit();
16 19         printfinit();
17 20
18 $1 = {void ()} 0x900000000200060 <main>
```

```
19 A debugging session is active.
20
21     Inferior 1 [process 1] will be detached.
22
23 Quit anyway? (y or n) [answered Y; input not from terminal]
24 [Inferior 1 (process 1) detached]
```

4.2 实验一：新增可执行程序测试

学生完成作业时，需要严格按照实验手册所规定的文件名，函数名，变量名，否则可能导致测评错误。

4.2.1 磁盘映像的生成

xv6上的可执行文件生成过程包含两个步骤：

- (1) 生成各个应用程序。
- (2) 将应用程序构成文件系统映像。

首先，在Makefile的变量UPROGS中添加可执行文件名my-app：

```
1 UPROGS=\
2 .....
3     $U/_usertests\
4     $U/_my-app\
5 #   $U/_grind\
6 .....
```

4.2.2 添加简单程序

在/usr/下编写一个my-app.c，作为用户程序：

```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4
5 int main(int argc, char* argv[])
6 {
7     printf("This is my own app!\n");
8     exit(0);
9 }
```

运行效果如下：

```
1 rex@rex-virtual-machine:~/xv6/xv6-loongarch-exp/qemu-loongarch-runenv$
  ./run_loongarch.sh -k ../kernel/kernel
2 .....
3 entry kernel ...
4 cpu0: starting xv6
5 init: starting sh
6 $ my-app
7 This is my own app!
8 $
```


4.2.3 验证可执行程序

综上，可以编写以下脚本测评用户是否添加了一个输出“This is my own app!”的可执行文件：

服务器端，启动qemu，运行xv6。

脚本server.sh如下：

```
1  ../../qemu-loongarch-runenv/qemu/x86_64/qemu-system-loongarch64\  
2  -m 1G -smp 1 -bios \  
3  ../../qemu-loongarch-runenv/loongarch_bios_0310_debug.bin\  
4  -kernel ../../kernel/kernel \  
5  -initrd ../../qemu-loongarch-runenv/busybox-rootfs.img \  
6  -append 'root=/dev/ram console=ttyS0,115200 rdinit=/init' \  
7  -vga none -nographic \  
8  -L ../../qemu-loongarch-runenv\  
9  -serial tcp::4556,server,nowait
```

客户端，连接qemu，向xv6发送“my-app”，若接受到“This is my own app!”，说明用户添加了可执行文件。

脚本client.sh如下：

```
1  echo 'my-app' | nc -w 1 127.0.0.1 4556 > test.output  
2  echo 'my-app' | nc -w 1 127.0.0.1 4556 > test.output  
3  
4  strA=$(cat test.output)  
5  strB='This is my own app!'  
6  result=$(echo ${strA} | grep "${strB}")  
7  
8  if [[ "${result}" != "" ]]  
9  then  
10     echo "The answer is right!"  
11 else  
12     echo "The answer is wrong!"  
13 fi  
14  
15 echo "The test is finished!"
```

4.3 实验二：新增系统调用的测试

4.3.1 添加系统调用

增加系统调用号

xv6 的系统调用都有一个唯一编号，定义在 kernel/syscall.h 中。我们可以在 SYS_close 的后面，新加入一行“#define SYS_getcpuid 22”即可，这里的编号 22 可以是其他值——只要不是前面使用过的就好。

修改syscall.h：

```
1  // system call numbers  
2  .....  
3  #define SYS_getcpuid 22  
4  .....
```

增加用户态入口

修改user.h, 添加“int getcpuid(void);”:

```
1 struct stat;
2 struct rtcdate;
3
4 // system calls
5 .....
6 int getcpuid(void);
7 .....
```

在user/usys.pl中定义用户态入口, 添加“entry("getcpuid");”:

```
1 #!/usr/bin/perl -w
2
3 # Generate usys.S, the stubs for syscalls.
4 .....
5 entry("uptime");
6 entry("getcpuid");
7 .....
```

修改syscall.c中的跳转表

在kernel/syscall.c中添加外部定义“extern uint64 sys_sh_var_read(void);”, 在跳转表中添加 “[SYS_getcpuid] sys_getcpuid,”。

```
1 .....
2 extern uint64 sys_uptime(void);
3 extern uint64 sys_getcpuid(void);
4 extern uint64 sys_sh_var_read(void);
5 .....
6 static uint64 (*syscalls[])(void) = {
7 .....
8 [SYS_close] sys_close,
9 [SYS_getcpuid] sys_getcpuid,
10 [SYS_sh_var_read] sys_sh_var_read,
11 .....
12 };
```

实现sys_getcpuid()

在sysproc.c中添加sys_getcpuid()的函数体:

```
1 .....
2 // return cpu id
3 uint64
4 sys_getcpuid(void)
5 {
6     return getcpuid();
7 }
8 .....
```

在proc.c中实现内核态的getcpuid():

```
1  .....
2  // get cpu id
3  int getcpuid()
4  {
5      int id = r_tp();
6
7      return id;
8  }
9  .....
```

为了让 sysproc.c 中的 sys_getcpuid()能调用 proc.c 中的 getcpuid(), 还需要在 kernel/defs.h加入一行 "int getcpuid(void);"

```
1  // proc.c
2  .....
3  void      procdump(void);
4  int       getcpuid(void);
5  void      wakeup1p(void*);
6  .....
```

4.3.2 添加可执行程序验证系统调用

添加如下的程序pcpuid.c, 调用上述添加的系统调用:

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int main(int argc, char* argv[])
6  {
7      printf("My cpu id is %d.\n", getcpuid());
8      exit(0);
9  }
```

运行结果如下:

```
1  rex@rex-virtual-machine:~/xv6/xv6-loongarch-exp/qemu-loongarch-runenv$
   ./run_loongarch.sh -k ../kernel/kernel
2  entry kernel ...
3  cpu0: starting xv6
4  init: starting sh
5  $ my-app
6  This is my own app!
7  $ pcpuid
8  My cpu id is 0.
9  $
```

4.3.3 使用shell脚本验证系统调用

综上，我们可以编写以下脚本验证系统调用。

服务器端，启动qemu，运行xv6。

脚本server.sh如下：

```
1  ../../qemu-loongarch-runenv/qemu/x86_64/qemu-system-loongarch64\  
2  -m 1G -smp 1 -bios \  
3  ../../qemu-loongarch-runenv/loongarch_bios_0310_debug.bin\  
4  -kernel ../../kernel/kernel \  
5  -initrd ../../qemu-loongarch-runenv/busybox-rootfs.img \  
6  -append 'root=/dev/ram console=ttyS0,115200 rdinit=/init' \  
7  -vga none -nographic \  
8  -L ../../qemu-loongarch-runenv\  
9  -serial tcp::4556,server,nowait
```

客户端，先检查用户是否有按要求添加系统调用，再使用可执行程序pcpuid验证其功能。

脚本client.sh如下：

```
1  str1=$(cat ../../kernel/syscall.h)  
2  test1="#define SYS_getcpuid"  
3  result1=$(echo ${str1} | grep "${test1}")  
4  
5  str2=$(cat ../../user/user.h)  
6  test2="int getcpuid(void);"  
7  result2=$(echo ${str2} | grep "${test2}")  
8  
9  str3=$(cat ../../user/usys.pl)  
10 test3='entry("getcpuid")'  
11 result3=$(echo ${str3} | grep "${test3}")  
12  
13 str4=$(cat ../../kernel/syscall.c)  
14 test4_1='extern uint64 sys_getcpuid(void);'  
15 result4_1=$(echo ${str4} | grep "${test4_1}")  
16  
17 test4_2='\[SYS_getcpuid\] sys_getcpuid'  
18 result4_2=$(echo ${str4} | grep "${test4_2}")  
19  
20 # echo ${result4_2}  
21 echo 'pcpuid' | nc -w 1 127.0.0.1 4556 > test.output  
22 echo 'pcpuid' | nc -w 1 127.0.0.1 4556 > test.output  
23  
24 strA=$(cat test.output)  
25 strB='My cpu id is 0.'  
26  
27 result=$(echo ${strA} | grep "${strB}")  
28  
29 if [[ "${result1}" != "" && "${result2}" != "" && "${result3}" != "" &&  
30     "${result4_1}" != "" && "${result4_2}" != "" && "${result}" != "" ]]  
31 then  
32     echo "The answer is right!"  
33 else  
34     echo "The answer is wrong!"
```

```

35  fi
36
37  echo "The test is finished!"

```

4.4 实验三：调整时间片长度

4.4.1 增加时间片信息

在 xv6 的进程控制块 kernel/proc.h 中修改 proc 结构体，增加成员 slot 并定义时间片长度为 8 个 tick。

```

1  struct proc
2  {
3      .....
4      char name[16];           // Process name (debugging)
5      int slot;                // time slot (ticks)
6      int priority;            // Process priority (0-20)=(highese-lowest)
7  };
8  #define SLOT 8

```

然后在 kernel/proc.c 中，创建进程时分配 proc 结构体的 allocproc() 函数中设置新进程的 slot 成员，并它设置为 SLOT。

```

1  .....
2  // Look in the process table for an UNUSED proc.
3  // If found, initialize state required to run in the kernel,
4  // and return with p->lock held.
5  // If there are no free procs, or a memory allocation fails, return 0.
6  static struct proc*
7  allocproc(void)
8  {
9      .....
10     found:
11         p->pid = allocpid();
12         p->state = USED;
13         p->slot = SLOT;
14         p->priority = 10; //default priority
15         .....
16     }

```

在 kernel/trap.c 中，检查时间片是否用完：

```

1  .....
2  //
3  // handle an interrupt, exception, or system call from user space.
4  // called from uservec.S
5  //
6  void
7  usertrap(void)
8  {
9      .....
10     if(p && p->state == RUNNING)
11     {
12         p->slot--;

```

```

13     if(p->slot == 0 )
14     {
15         p->slot = 8;
16         yield();
17     }
18 }
19
20 usertrapret();
21 }

```

4.4.2 添加可执行程序查看时间片长度变化

添加如下的程序loop.c, 查看进程时间片的变化:

```

1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int
6  main(int argc, char *argv[])
7  {
8      int pid;
9      int data[8];
10     int i, j, k;
11
12     pid=fork();
13     for( i = 0; i < 2; i ++){
14         {
15             for( j= 0; j < 1024 * 100; j ++){
16                 for( k = 0; k < 1024 * 1024; k ++){
17                     data[k % 8] = pid * k;
18                 }
19                 printf("%d ", data[0]);
20                 exit(0);
21             }

```

4.4.3 使用shell脚本验证时间片长度的变化

综上, 我们可以编写以下脚本验证系统调用:

服务器端, 启动qemu, 运行xv6。

脚本server.sh如下:

```

1  ../../qemu-loongarch-runenv/qemu/x86_64/qemu-system-loongarch64\
2  -m 1G -smp 1 -bios \
3  ../../qemu-loongarch-runenv/loongarch_bios_0310_debug.bin \
4  -kernel ../../kernel/kernel \
5  -initrd ../../qemu-loongarch-runenv/busybox-rootfs.img \
6  -append 'root=/dev/ram console=ttyS0,115200 rdinit=/init'\
7  -vga none -nographic \
8  -L ../../qemu-loongarch-runenv \
9  -serial tcp::4556,server,nowait

```

客户端, 检查用户是否有按照要求修改内核代码。

脚本client.sh如下:

```
1  str1_1=$(cat ../../kernel/proc.h)
2  test1_1="int slot;"
3  result1_1=$(echo ${str1_1} | grep "${test1_1}")
4
5  str1_2=$(cat ../../kernel/proc.h)
6  test1_2="#define SLOT 8"
7  result1_2=$(echo ${str1_2} | grep "${test1_2}")
8
9  str2_1=$(cat ../../kernel/proc.c)
10 test2_1="p->slot = SLOT;"
11 result2_1=$(echo ${str2_1} | grep "${test2_1}")
12
13 str2_2=$(cat ../../kernel/proc.c)
14 test2_2='printf("slice left:%d ticks,%d %s %s", p->slot, p->pid, state, p->name);'
15 result2_2=$(echo ${str2_2} | grep "${test2_2}")
16
17 if [[ "${result1_1}" != "" && ${result1_2} != "" && ${result2_1} != "" &&
    ${result2_2} != "" ]]
18 then
19     echo "The answer is right!"
20 else
21     echo "The answer is wrong!"
22 fi
23
24 echo "The test is finished!"
```

4.5 实验四：实现信号量

4.5.1 共享变量及其访问

共享变量

定义一个名为sh_var_for_sem_demo 的全局变量，在kernel中新建一个sem.h:

```
1  #include "types.h"
2
3  uint sh_var_for_sem_demo;
```

访问共享变量

新增两个系统调用sys_sh_rw_read()和sys_sh_rw_write(), 前者可以读取上述定义的共享变量sh_var_for_sem_demo, 后者可以修改上述定义的共享变量sh_var_for_sem_demo。

系统调用的添加过程与上述相似，不再赘述。

在sysproc.c中添加这两个系统调用的函数体:

```
1  .....
2  uint64
3  sys_sh_var_read()
4  {
5      return (uint64)sh_var_for_sem_demo;
```

```

6  }
7
8  uint64
9  sys_sh_var_write()
10 {
11     int n;
12     if(argint(0, &n) < 0)
13         return (uint64)-1;
14
15     sh_var_for_sem_demo = n;
16     return (uint64)sh_var_for_sem_demo;
17 }
18 .....

```

4.5.2 信号量数据结构

在kernel/spinlock.h 中，新增信号量数据结构的声明：

```

1  .....
2  #define SEM_MAX_NUM 128 //信号量总数
3  extern int sem_used_count; //当前在用信号量数目
4  struct sem
5  {
6      struct spinlock lock; //内核自旋锁
7      int resource_count; //资源计数
8
9      int allocated; //是否被分配使用：1 已分配，0 未分配
10 };
11 extern struct sem sems[SEM_MAX_NUM]; //系统可有 SEM_MAX_NUM 个信号量

```

4.5.3 信号量操作的系统调用

信号量操作涉及到以下4个系统调用：

- (1) int sem_create(int n_sem) 参数 n_sem 是初值，返回的是信号量的编号，-1 为出错
- (2) int sem_p(int sem_id) 减一操作，减为 0 时阻塞睡眠，记录到 sem.procs[] 中。返回值 0 表示正常，返回值-1 则出错。
- (3) int sem_v(int sem_id) 增一操作，增加到 0 时唤醒队列中的进程，清除sems[id].procs[]对应的进程号。返回值为 0 表示成功，-1 表示出错。
- (4) int sem_free(int sem_id) 释放指定 id 的信号量。返回值为 0 表示成功，-1 表示出错。

在sysproc.c中添加上述系统调用的代码：

```

1  .....
2  int sys_sem_create()
3  {
4      int n_sem, i;
5      if(argint(0, &n_sem) < 0 )
6          return -1;
7      for(i = 0; i < SEM_MAX_NUM; i++)
8      {
9          acquire(&sems[i].lock);
10         if(sems[i].allocated == 0)
11         {

```



```

12     sems[i].allocated = 1;
13     sems[i].resource_count = n_sem;
14     printf("create %d sem\n",i);
15     release(&sems[i].lock);
16     return i;
17 }
18 release(&sems[i].lock);
19 }
20 return -1;
21 }
22
23 int sys_sem_free()
24 {
25     int id;
26     if(argint(0, &id) < 0)
27         return -1;
28     acquire(&sems[id].lock);
29     if(sems[id].allocated == 1 &&
30        sems[id].resource_count > 0)
31     {
32         sems[id].allocated = 0;
33         printf("free %d sem\n", id);
34     }
35     release(&sems[id].lock);
36     return 0;
37 }
38
39 int sys_sem_p()
40 {
41     int id;
42     if(argint(0, &id) < 0)
43         return -1;
44     acquire(&sems[id].lock);
45     sems[id].resource_count--;
46     if(sems[id].resource_count<0) //首次进入、或被唤醒时，资源不足
47         sleep(&sems[id],&sems[id].lock); //睡眠（会释放 sems[id].lock 才阻塞）
48     release(&sems[id].lock); //解锁（唤醒到此处时，重新持有 sems[id].lock）
49     return 0; //此时获得信号量资源
50 }
51
52 int sys_sem_v(int sem_id)
53 {
54     int id;
55     if(argint(0,&id)<0)
56         return -1;
57     acquire(&sems[id].lock);
58     sems[id].resource_count+=1; //增 1
59     if(sems[id].resource_count<1) //有阻塞等待该资源的进程
60         wakeup1p(&sems[id]); //唤醒等待该资源的 1 个进程
61     release(&sems[id].lock); //释放锁
62     return 0;
63 }
64 .....

```

修正wakeup操作

在kernel/proc.c中添加wakeup1p()函数:

```
1  .....
2  void wakeup1p(void *chan)
3  {
4      struct proc *p;
5
6      for (p = proc; p < &proc[NPROC]; p++)
7      {
8          if(p != myproc())
9          {
10             acquire(&p->lock);
11             if(p->state == SLEEPING && p->chan == chan)
12             {
13                 p->state = RUNNABLE;
14                 release(&p->lock);
15                 break;
16             }
17             release(&p->lock);
18         }
19     }
20 }
```

4.5.4 添加可执行程序验证信号量

添加如下的程序sh_rw_lock.c, 调用上述添加的系统调用:

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int main()
6  {
7      int id=sem_create(1);
8      int pid = fork();
9      int i;
10     for(i=0;i<1000;i++)
11     {
12         sem_p(id);
13         sh_var_write(sh_var_read()+1);
14         sem_v(id);
15     }
16     if(pid > 0)
17     {
18         wait(0);
19         sem_free(id);
20     }
21     printf("sum=%d\n", sh_var_read());
22     exit(0);
23 }
```

运行结果如下:

```

1 rex@rex-virtual-machine:~/xv6/xv6-loongarch-exp/qemu-loongarch-runenv$
  ./run_loongarch.sh -k ../kernel/kernel
2 entry kernel ...
3 cpu0: starting xv6
4 init: starting sh
5 $ sh_rw_lock
6 create 0 sem
7 sum=2000
8 free 0 sem
9 sum=2000
10 $

```

4.5.5 使用shell脚本验证信号量

综上，可以编写以下shell脚本测试信号量的实现。

综上，我们可以编写以下脚本验证系统调用：

服务器端，启动qemu，运行xv6。

脚本server.sh如下：

```

1 ../../qemu-loongarch-runenv/qemu/x86_64/qemu-system-loongarch64\
2 -m 1G -smp 1 -bios \
3 ../../qemu-loongarch-runenv/loongarch_bios_0310_debug.bin\
4 -kernel ../../kernel/kernel \
5 -initrd ../../qemu-loongarch-runenv/busybox-rootfs.img \
6 -append 'root=/dev/ram console=ttyS0,115200 rdinit=/init' \
7 -vga none -nographic \
8 -L ../../qemu-loongarch-runenv\
9 -serial tcp::4556,server

```

客户端，使用可执行程序sh_rw_lock验证其功能。

脚本client.sh如下：

```

1 echo 'sh_rw_lock' | nc -w 2 127.0.0.1 4556 > test.output
2 echo 'sh_rw_lock' | nc -w 2 127.0.0.1 4556 > test.output
3
4 strA=$(cat test.output)
5 strB='sum=2000'
6 result=$(echo ${strA} | grep "${strB}")
7
8 if [[ "${result}" != "" ]]
9 then
10     echo "The answer is right!"
11 else
12     echo "The answer is wrong!"
13 fi
14
15 echo "The test is finished!"

```

五、遇到的问题和解决方法

5.1 如何使脚本能够和xv6通信

最开始的时候，不知道如何让脚本和xv6之间进行通信，求助了黄进科学长，得知了可以使用qemu的serial参数将xv6串口重定向到tcp端口，因此可以通过脚本或程序与xv6的终端进行互操作，从而为实现自动评判提供辅助。

具体的实现见“4.1 调试与观察手段”。

5.2 关于脚本的编写

关于本实验，使用python应当是更好的选择，但队员都不擅长python，于是选择了编写shell脚本，有一定的局限性，如可读性一般、可移植性较差。

六、系统测试情况

6.1 测试之前的系统准备

首先克隆远程的初试仓库：

```
1 git clone https://gitlab.eduxiji.net/202310590111598/project1466467-177157.git
```

获取远程仓库所有分支执行下面的脚本：

```
1 git branch -r | grep -v '\->' | while read remote; do git branch --track  
  "${remote#origin/}" "$remote"; done  
2 git fetch --all  
3 git pull --all
```

接着来到分支Ex1(默认分支)，可以使用git checkout branchName来切换到对应分支，查看所有分支情况可使用git branch -a，来到分支Ex1下之后，执行下面命令：

```
1 cd qemu-loongarch-runenv # 切换到qemu-loongarch-runenv  
2 git submodule update --init # 子模块初始化，获取远程仓库对应的运行环境文件
```

接着配好对应的交叉编译环境，具体参考本项目的README.md文档中的xv6-loongarch的安装使用，接着在项目的根目录下执行下面命令(可能需要root权限)：

```
1 make all # 编译xv6内核
```

完成xv6内核的编译。

6.2 判题系统测试步骤

6.2.1 本地判题系统测试

按照实验手册的要求，在对应Ex分支中添加修改对应的文件之后，重新编译xv6内核，即在项目根目录中执行make all。

接着执行下面的命令：

```
1 cd Test
2 ./server.sh # 打开终端执行
3 ./client.sh # 打开另一个终端执行
```

注：上述的两个不同脚本分别需要在两个不同终端上执行。

6.2.2 远程判题系统测试(未完成，待续)

按照实验手册的要求，在**对应Ex分支**中添加修改对应的文件之后，重新编译xv6内核，即在项目根目录中执行 `make all`。

学生客户端

使用 `git diff` 命令生成相应的patch文件，在作业网页对应实验页面上提交patch，生成patch的相应命令如下：

```
1 git add -u # 添加所有修改到暂存区中
2 git reset -- kernel/ramdisk.h # 排除执行make all而添加的修改文件
3 git diff --cached > commit.patch # 将暂存区修改和当前版本库进行对比
```

将根目录生成的 `commit.patch` 文件上传到服务器上。

判题服务端

服务器通过网络接收到对应学生提交过来的 `commit.patch` 文件，接着将对应的文件和本地配置好的项目进行**打补丁**操作，来添加学生的修改，将 `commit.patch` 添加到项目根目录下，执行下面命令：

```
1 git apply --check commit.patch # 检测patch是否可以应用，没有报错输出证明可用
2 git apply commit.patch # 使用该命令来打补丁
```

完成打补丁操作之后，远程服务器模仿本地判题系统测试方法，一次执行对应命令进行判题。

注：远程判题系统测试预想中是需要实现**自动评测**功能的，但是由于时间有限和小组中成员没有开发过相应的前后台系统，暂时没有做出对应的提交系统文件UI前端界面和docker判题服务器。

6.3 判题系统可行性验证——以实验一为例

6.3.1 判题系统可行性验证思路

模拟学生角色，先在一个空闲目录中克隆远程仓库，完成实验一的my-app应用程序添加，然后依照上述的2中提到的**学生客户端**生成patch的步骤，生成相应的 `commit.patch` 补丁包。

接着，模拟远程提交补丁包，将 `commit.patch` 补丁包添加到另一个配好环境生成kernel的可执行xv6仓库根目录中，依照上述2中提到的**判题服务端**打补丁自动判题的步骤，手动执行命令并观察判题结果。

6.3.2 判题系统验证

模拟学生客户端操作

根据实验指导添加my-app用户态应用程序，接着修改 `Makefile` 添加改应用程序的文件映像，重新编译之后，使用 `git status` 查看当前仓库下的修改情况，如下：

```

no changes added to commit (use "git add" and/or "git commit -a")
● kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git status
On branch Ex1
Your branch is up to date with 'origin/Ex1'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Makefile
        modified:   kernel/ramdisk.h

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        user/my-app.c

no changes added to commit (use "git add" and/or "git commit -a")

```

将上述修改文件添加到git暂存区中，如下：

```

● kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git add Makefile user/my-app.c
● kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git status
On branch Ex1
Your branch is up to date with 'origin/Ex1'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Makefile
        new file:   user/my-app.c

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   kernel/ramdisk.h

```

注意：`kernel/ramdisk.h` 修改是编译生成的，不是手动添加修改，而使用 `git diff` 是要生成手动修改前后文件不同。

将修改文件添加到版本库，生成新的commit，如下：

```

● kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git commit -m '增加应用my-app'
[Ex1 d1eb819] 增加应用my-app
2 files changed, 11 insertions(+)
create mode 100644 user/my-app.c

```

通过不同commit ID作为参数，使用 `git diff` 生成对应的 `commit.patch`，如下：

```

kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git log
commit d1eb8192da185bff45c4fa5b100f992d5a4f3e4f (HEAD -> Ex1)
Author: Kitebin-h <kitebin01@gmail.com>
Date: Wed Jun 7 05:07:23 2023 +0800

    增加应用my-app

commit 8a89910bfbf5a0ed65b8641cd822cc5fbeb82724 (origin/HEAD, origin/Ex1)
Author: Kitebin-h <kitebin01@gmail.com>
Date: Wed Jun 7 02:10:56 2023 +0800

    update Test

commit bd2c1bca909b06d3237d73081fd5566620d8920a
Author: Kitebin-h <kitebin01@gmail.com>
Date: Tue Jun 6 22:35:43 2023 +0800

    update Test

commit d19e6ed948eb584ec7c9302c4bb9cd6359584ff5
Author: Kitebin-h <kitebin01@gmail.com>
Date: Tue Jun 6 22:26:17 2023 +0800

    update Test

commit 7cc5138070f27286a8f9cb77166ef782765ce372
Author: Kitebin-h <kitebin01@gmail.com>
Date: Tue Jun 6 22:11:48 2023 +0800

    update Test
kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ ^C
kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git diff ^C
kitebin@LAPTOP-70E651IU:~/project/project1466467-177157$ git diff 8a89910bfbf5a0ed65b8641cd
822cc5fbeb82724 d1eb8192 > commit.patch

```

查看 `commit.patch` 中的内容:

```

C my-app.c  M Makefile  commit.patch U X
commit.patch
1 diff --git a/Makefile b/Makefile
2 index da00ddb..1e389e0 100644
3 --- a/Makefile
4 +++ b/Makefile
5 @@ -117,6 +117,7 @@ UPROGS=\
6     $U/_sh\
7     $U/_stressfs\
8     $U/_usertests\
9 +    $U/_my-app\
10    # $U/_grind\
11    $U/_wc\
12    $U/_zombie\
13 diff --git a/user/my-app.c b/user/my-app.c
14 new file mode 100644
15 index 0000000..48a139c
16 --- /dev/null
17 +++ b/user/my-app.c
18 @@ -0,0 +1,10 @@
19 +#include "kernel/types.h"
20 +#include "kernel/stat.h"
21 +#include "user/user.h"
22 +
23 +int
24 +main(int argc, char* argv[]) {
25 +
26 +    printf("This is my own app!\n");
27 +    exit(0);
28 +}
29 \ No newline at end of file
30

```

本地添加 `my-app` 应用程序后编译运行测试:

```
kitebin@LAPTOP-70E651IU:~/project/project1466467-177157/Test$ ./server.sh
QEMU 6.2.50 monitor - type 'help' for more information
(qemu) qemu-system-loongarch64: terminating on signal 2
kitebin@LAPTOP-70E651IU:~/project/project1466467-177157/Test$ ./server.sh
kitebin@LAPTOP-70E651IU:~/project/project1466467-177157/Test$ ./client.sh
The answer is right!
The test is finished!
```

可以看到新添加的 `my-app` 能够通过判题系统本地测试。

模拟评测服务端操作

将客户端生成的 `commit.patch` 文件添加到服务端的根目录下, 使用 `git status` 查看:

```
kitebin@LAPTOP-70E651IU:~/xv6-automated-scoring-system/project1466467-177157$ git status
On branch Ex1
Your branch is ahead of 'origin/Ex1' by 1 commit.
(use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    commit.patch

nothing added to commit but untracked files present (use "git add" to track)
```

使用命令 `git apply --check commit.patch` 检查补丁包是否可用, 如下:

```
kitebin@LAPTOP-70E651IU:~/xv6-automated-scoring-system/project1466467-177157$ git apply --check
k commit.patch
kitebin@LAPTOP-70E651IU:~/xv6-automated-scoring-system/project1466467-177157$ git apply commit
```

可以看到检查补丁包没有报错, 正常可用。

应用 `commit.patch` 对原系统进行打补丁:

```
kitebin@LAPTOP-70E651IU:~/xv6-automated-scoring-system/project1466467-177157$ git apply commit.patch
commit.patch:23: trailing whitespace.
int
warning: 1 line adds whitespace errors.
```

打补丁之后重新编译运行xv6系统, 执行Test文件夹中相应的脚本对评测系统添加功能进行功能评测。

```
kitebin@LAPTOP-70E651IU:~/xv6-automated-scoring-system/project1466467-177157/Test$ ls
client.sh  server.sh
kitebin@LAPTOP-70E651IU:~/xv6-automated-scoring-system/project1466467-177157/Test$ ./client.sh
The answer is right!
The test is finished!
kitebin@LAPTOP-70E651IU:~/xv6-automated-scoring-system/project1466467-177157/Test$
```

可以看到评测结果和客户端一致。

重新编译运行之后, 查看本地更改文件

```

  6
  U commit.patch
  M Makefile
  M ramdisk.h kernel
  M server.sh Test
  U test.output Test
  U my-app.c user
```

各个文件更改情况:

- `my-app.c`: 手动添加的用户态应用程序, 工作树修改如下:


```
> C my-app.c > ...
1-
1+ #include "kernel/types.h"
2+ #include "kernel/stat.h"
3+ #include "user/user.h"
4+
5+ int
6+ main(int argc, char* argv[]) {
7+
8+     printf("This is my own app!\n");
9+     exit(0);
10+ }
```

- **Makefile**: 添加文件映像, 工作树修改如下:

```
106 UPROGS=\
107     $U/_cat\
108     $U/_echo\
109     $U/_forktest\
110     $U/_grep\
111     $U/_init\
112     $U/_kill\
113     $U/_ln\
114     $U/_ls\
115     $U/_mkdir\
116     $U/_rm\
117     $U/_sh\
118     $U/_stressfs\
119     $U/_usertests\
120 #   $U/_grind\
121     $U/_wc\
122     $U/_zombie\
123
124 fs.img: mkfs/mkfs README $(UPROGS)
125     mkfs/mkfs fs.img README $(UPROGS)
126     xxd -i fs.img > kernel/ramdisk.h
127
128
106 UPROGS=\
107     $U/_cat\
108     $U/_echo\
109     $U/_forktest\
110     $U/_grep\
111     $U/_init\
112     $U/_kill\
113     $U/_ln\
114     $U/_ls\
115     $U/_mkdir\
116     $U/_rm\
117     $U/_sh\
118     $U/_stressfs\
119     $U/_usertests\
120+    $U/_my-app\
121 #   $U/_grind\
122     $U/_wc\
123     $U/_zombie\
124
125 fs.img: mkfs/mkfs README $(UPROGS)
126     mkfs/mkfs fs.img README $(UPROGS)
127     xxd -i fs.img > kernel/ramdisk.h
128
```

- **test.output**: 输出文本重定向文件, 工作树修改如下:

```
test.output
1-
1+ my-app
2+ This is my own app!
3+ $
```

- **ramdisk.h**: 重新编译修改的文件
- **server.sh**: 手动修改启动路径, 与实验无关
- **commit.patch**: 根目录下添加的patch文件

七、总结和展望

本次初赛, 我们初步对于一开始判题系统设计设想进行验证其可行性, 选用**龙芯xv6操作系统实验**中四个小实验分别进行测试, 结果说明方案的可行性。由于小组成员都是大三生, 在兼顾课业之余抽出时间来进行比赛, 时间有限能力不足, 暂未完善远程评测系统的自动化评测功能, 希望后面有机会能好好完善该自动评测系统。

对于未来评测系统设计的设想:

- 使用Python脚本重写判题部分的脚本, 使其具备通用性(本次实验使用bash脚本编写, 可能不适用与其他终端)。
- 添加更多实验自动评测系统, 脚本中添加更严格的测试案例和数据。
- 构建远程评测系统docker服务器, 并且设计前端UI提交patch学生web客户端, 学生根据学号密码登录web客户端, 在对应网页提交对应实验生成的patch, 远程docker服务器接收patch重新构建编译教学系统, 进行评测机评测并把对应的结果返回给学生web客户端网页。

八、分工协作

- 黄万丰: 负责四个小实验评测代码shell脚本编写, 部分文档编写工作。
- 黄灿彬: 负责代码协同管理, 将shell脚本添加到版本控制的四个分支中并进行逐一分支**本地**测试, 使用 **git diff** 生成补丁(patch)和打补丁的方式模拟远程服务端实验测试, 部分文档编写工作。

九、提交仓库目录和文件描述

仓库目录结构如下:

```
1  .
2  |— Makefile
3  |— README
4  |— README.md
5  |— imgs
6  |— Test
7  |   |— client.sh
8  |   └─ server.sh
9  |— kernel
10 |   |— apic.c
11 |   |— apic.d
12 |   ...
13 |   |— vm.c
14 |   |— vm.d
15 |   └─ vm.o
16 |— mkfs
17 |   |— mkfs
18 |   └─ mkfs.c
19 |— qemu-loongarch-runenv
20 |— test.sh
21 └─ user
    |— _cat
    ...
    |— usys.pl
    |— wc.c
    └─ zombie.c
```

Makefile : 编写编译 **xv6** 内核的编译语句

README : **xv6** 操作系统介绍文档

README.md : 项目介绍文档

imgs : **LoongArch** 架构上可运行的 **xv6** 内核运行图片

Test : 存放操作系统实验判题脚本

- **server.sh** : 服务端脚本负责启动 **qemu** , 并将输出重定向到某个TCP端口上
- **client.sh** : 客户端脚本负责连接服务器TCP端口, 通过命令与 **qemu** 中仿真的 **xv6** 进行交互

kernel : 用来存放操作系统内核代码

mkfs : 用来存放构建文件镜像实现代码

qemu-loongarch-runenv : 包括启动 **qemu** 所需的文件, 这里是子模块, 克隆到本地需要先初始化文件夹

test.sh : 用户运行 **qemu** 启动 **xv6** 仿真测试脚本

user : 用来存放用户程序, 其中包括函数库, 系统调用和用户程序实现等

十、比赛收获

- 熟悉git进行协同代码开发流程，学习冲突处理
- 熟悉了shell脚本的编写
- 深入理解xv6实验以及判题思路
- 提高自己对于系统设计能力
- 进行了判题系统方案的可行性的初步验证