

lab3 report

调度器

任课教师：叶保留 助教：尹熙喆，水兵

| | | | |
|-------|--|---------|---------------------|
| 学院 | 计算机科学与技术系 | 专业 | 计算机科学与技术 |
| 学号 | 201220154 | 姓名 | 王紫萁 |
| Email | 201220154@smail.nju.edu.cn | 开始/完成时间 | 2022. 4.7/2022.4.15 |

1. exercises

• exercise 1

exercise1: 请把上面的程序，用gcc编译，在你的Linux上面运行，看看真实结果是啥（有一些细节需要改，请根据实际情况进行更改）。

```
1 | Father Process [449]: Ping 1, 7;
2 | Child Process [450]: Pong 2, 7;
3 | Father Process [449]: Ping 1, 6;
4 | Child Process [450]: Pong 2, 6;
5 | Father Process [449]: Ping 1, 5;
6 | Child Process [450]: Pong 2, 5;
7 | Father Process [449]: Ping 1, 4;
8 | Child Process [450]: Pong 2, 4;
9 | Child Process [450]: Pong 2, 3;
10 | Father Process [449]: Ping 1, 3;
11 | Child Process [450]: Pong 2, 2;
12 | Father Process [449]: Ping 1, 2;
13 | Father Process [449]: Ping 1, 1;
14 | Child Process [450]: Pong 2, 1;
15 | Child Process [450]: Pong 2, 0;
16 | Father Process [449]: Ping 1, 0;
```

其中方括号内显示了当前的进程号。 `fork` 函数会拷贝一份当前进程的副本作为当前进程的子进程，并在子进程中返回0，父进程中返回子进程的 `pid`，随后两个进程都从 `fork` 开始继续执行。并由操作系统负责决定之后父子进程的执行顺序。

• exercise 2

- exercise2: 请简单说说, 如果我们想做虚拟内存管理, 可以如何进行设计 (比如哪些数据结构, 如何管理内存) ?

本次实验采取段级地址的内存转换, 对不同的进程只需要改变段选择子的值就能定位到进程地址空间。Linux中采用段页式虚存管理, 将48位逻辑地址首先根据段选择子和GDTR寄存器确定线性地址 (查找 GDT), 再根据页目录表 (PDT) 和页表 (PT) 找到物理地址。分配内存时, 通过记录内存中空闲的页框将用户程序加载到内存的空闲位置上。

• exercise 3

- exercise3: 我们考虑这样一个问题: 假如我们的系统中预留了100个进程, 系统中运行了50个进程, 其中一些结束了运行。这时我们又有一个进程想要开始运行 (即需要分配给它一个空闲的PCB), 那么如何能够以 $O(1)$ 的时间和 $O(n)$ 的空间快速地找到一个空闲PCB呢?

$O(1)$ 时间 $O(n)$ 空间的查找可以采用一个记录空闲 PID 的栈, 每次需要分配空闲PID时就弹栈, 初始状态下链表中存放了100个进程的进程号均为空闲。当有 PCB 转为空闲时将它入栈即可。在**本次实验中**我们实现了一个基于这种逻辑的简单查找并提供 `getValidPid()` 和 `putValidPid(uint32_t pid)` 两个接口, 分别在 `fork` 和 `exit` 时显式调用得到或放入空闲 PCB

• exercise 4

- exercise4: 请你说说, 为什么不同用户进程需要对应不同的内核堆栈?

因为不同进程的内核栈内容不同, 可以举个不成立的情形: 如果两个用户进程使用同一个栈, 那么在需要进程调度时, 由进程A转换到进程B, A需要在内核栈保存自己的状态信息, 而B需要从内核栈取回状态信息, A压栈后B取回的就是A的状态信息, 显然这是不合适的, 更不用说当有多个进程需要来回切换时, 并不明确取回的信息到底是不是自己的。

• exercise 5

- exercise5: `stackTop`有什么用? 为什么一些地方要取地址赋值给`stackTop`?

从 `irqHandle` 我们看到 `stackTop` 保存了进程当前的 `StackFrame` (用户级的 `StackFrame` (在用户程序的内核栈), 其中有中断号等信息) (中断嵌套的情况), 从中断返回或者切换进程时, 当前进程的`stackTop`被指向了待执行用户进程的 `StackFrame`, 通过将 `stackFrame` 中的成员逐个弹出之后 `iret` 返回到上一次中断或者当前进程执行

• exercise 6

- exercise6: 请说说在中断嵌套发生时, 系统是如何运行的?

一个进程正在在内核态处理中断的过程中有另一个中断到来, 硬件自动将 `eflags`, `cs`, `eip` 入栈后进入到 `irq_Handle`, 此时又更新了 `stackTop` 的值, 使其保存当前状态的 `sf` 以便执行新来的中断, 于是内核栈中就会有服务两个中断的 `StackFrame`, 当从中断返回时, `StackFrame` 及其首地址都被弹出继续执行上一次中断的服务例程。

• exercise 7

- exercise7: 那么, 线程为什么要以函数为粒度来执行? (想一想, 更大的粒度是....., 更小的粒度是.....)

比函数更小的粒度是指令 (或者一条高级语言的语句其中包含多个指令但不能任意多), 更大的粒度就是进程了。函数可以看作执行一个进程需要完成的小任务, 这与线程存在的意义正好符合, 单一的指令可以由CPU管理, 一个完整任务的执行可以由进程来管理, 为了完成整个任务需要完成的小任务 (函数) 由线程管理。

• exercise 8

- exercise8: 请用fork, sleep, exit自行编写一些并发程序, 运行一下, 贴上你的截图。(自行理解, 不用解释含义, 帮助理解这三个函数)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5
6  int main() {
7      printf("%d: main process\n", getpid());
8      int rc = fork();
9      if (rc == 0) {
10         printf("%d: child process, i'm gonna execute a new program\n",
getpid());
11         char* command = "./exec 1st 2ed";
12         system(command);
13         exit(1);    //never reach here
14     } else if (rc > 0) {
15         sleep(1);
16         printf("%d: i'm father process\n", getpid());
17         exit(0);
18     }
19 }
20
21 //
22 // exec.c
23 //
24 #include <stdio.h>
25 int main(int argc, char** args) {
26     printf("i am another process and your arguments are as follows\n");
27     for (int i = 1; i < argc; ++i) {
28         printf("%s\n", args[i]);
29     }
30 }
```

运行结果如下, 值得一提的是, system函数内部有一条execl指令用于新建一个shell进程并把当前待执行的命令输入到shell中, 于是也就可以执行两个不同可执行文件了。

```
4428: main process
4429: child process, i'm gonna execute a new program
i am another process and your arguments are as follows
1st
2ed
4428: i'm father process
```

• exercise 9

- exercise9: 请问，我使用loadelf把程序装载到当前用户程序的地址空间，那不会把我loadelf函数的代码覆盖掉吗？（很傻的问题，但是容易产生疑惑）

从执行结果我们看到并不会覆盖，loadelf 属于内核级别的代码，处于 0x100000 的地址空间，而我们的用户程序是从 0x200000 以后开始的，因此不会产生覆盖冲突。

2. challenges

• challenge 2

- challenge2: 请说说内核级线程库中的 pthread_create 是如何实现即可。

```
1 int
2 __pthread_create_2_1 (pthread_t *newthread, const pthread_attr_t *attr,
3                       void *(*start_routine) (void *), void *arg)
```

对照源码我们解释一下基本原理：

- 在linux中有一个特殊系统级线程 __pthread_manager_thread 用于管理所有的线程，创建线程时会向该线程发送CREATE请求，并将新线程的函数入口和参数等信息也一并打包发送
- 该系统级线程收到创建线程的请求后会进行信号初始化，分配内存和栈空间等信息
- 执行 pthread_handle_create 其中会执行 __clone 函数并根据预处理的属性值等创建一个新的线程栈空间，和父线程共享同一地址空间，可以修改和读取。
- 重启进程并进行创建过程中用掉的栈空间的回收工作，检查是否成功创建后将该进程控制信息加入到线程管理的双向链表同一管理。

• challenge 3

- challenge3: 你是否能够在框架代码中实现一个简易内核级线程库，只需要实现create, destroy函数即可，并仍然通过时钟中断进行调度，并编写简易程序测试。不写不扣分，写出来本次实验直接满分。

抱着尝试的心态，实现了 pthread_create 和 pthread_destroy 函数能够让进程调用这两个函数创建自己并发的线程。大致思路是，每个进程控制块 PCB 的结构需要简单改一下，并增加 TCB 结构体作为线程控制块，每个TCB有自己的内核栈，在时钟中断到来时，会首先检查当前进程的状态以及时间片是否耗尽等信息。如果当前进程处于 RUNNING 状态并且还有空闲时间片，我们检查该进程下的所有线程信息（每个线程在创建开始时都有一个初始线程放在 tcb[0] 通常是主函数的线程。但是在我们的实现中**并没有刻意强调线程的父子关系**，即所有线程在执行地位上是**平等的**。）TCB, PCB 的重新定义如下：

```
1 struct ThreadTable {
2     uint32_t stack[MAX_STACK_SIZE];
3     struct StackFrame regs;
4     uint32_t stackTop;
5     int state; //有效状态为: STATE_DEAD, STATE_RUNNING, STATE_RANNABLE
6     int timeCount;
7     int sleepTime;
8     uint32_t pid;
9 };
10 typedef struct ThreadTable ThreadTable;
11
```

```

12 struct ProcessTable {
13     ThreadTable tcb[MAX_PCB_NUM];    //线程结构体
14     int state;
15     int timeCount;
16     int sleepTime;
17     uint32_t pid;
18     uint32_t currentTid;    //进程执行的当前线程
19     char name[32];
20 };
21 typedef struct ProcessTable ProcessTable;

```

当当前线程的时间片耗尽时，查找有没有RUNNABLE状态的线程，如果有就调用 `switch_thr` 函数进行状态转化，具体而言我们需要修改执行线程为目标线程并进行一系列恢复寄存器状态的操作后 `iret` 返回用户态，和进程切换的过程类似。否则继续执行当前线程。（当前线程的线程号记录在 PCB 结构体的 `currentTid` 成员中）

如果当前进程的状态为 `DEAD` 我们会尝试切换到一个RUNNABLE的线程，如果没有找到我们会在下一个时钟中断到来时处理这种情况（即没有线程正在执行，需要让当前进程消亡，下一段会具体介绍）；

因为所有线程的地位是平等的，因此不会像进程一样当无法找到可以执行的线程时就转到0号系统线程执行，而由于线程之间的并发关系我们也不能在0号线程（主线程）中用 `exit` 退出，否则其他线程得不到执行的机会，因此所有的线程在结束时都会调用 `pthread_destroy` 销毁自己。系统会在每个时钟中断到来时检查当前进程是否有正在执行或者可以执行的线程，如果没有意味着当前进程可以安全地释放。

容易出错的点在于，如果同一进程的所有线程共享同一用户栈，因此和进程切换不同的是当有中断到来打断当前线程执行时，`int`指令会将SS, ESP, EFLAGS, CS, EIP信息全部入栈，切换线程后，准备返回时从栈中取得的还是上一个线程的三个寄存器的信息。

解决的方法是：每个线程应该使用一个独立的用户栈用于实现自己的栈帧维护。实现的方法为在创建线程时需要把TCB中的用户现场信息regs中的ESP做分配处理 `esp = 0x100000 - 0x10000 * tid`。这里我们给每个线程划分0x10000大小的栈空间，每个用户栈在逻辑内存中如下静态划分

| | | |
|----|----------|-------------------|
| 1 | 0x100000 | +-----+ |
| 2 | | tcb[0] user stack |
| 3 | 0x0f0000 | +-----+ |
| 4 | | tcb[1] user stack |
| 5 | 0x0e0000 | +-----+ |
| 6 | | tcb[2] user stack |
| 7 | 0x0d0000 | +-----+ |
| 8 | | tcb[3] user stack |
| 9 | 0x0c0000 | +-----+ |
| 10 | . | |
| 11 | . | HEAP and segments |
| 12 | . | |
| 13 | 0x000000 | +-----+ |
| 14 | | |

我们用两个测试用例检验实现的效果（为了能够切换，我们每到来一个时钟中断就切换一次，i.e. `MAX_TIME_COUNT = 1`）

1. 不访问全局变量

删除一部分app_print的打印过程，并创建如下两个线程函数

```

1 void t1() {
2     for (int cnt = 0; cnt < 15; cnt++) {
3         printf("thread p1, loop: %d\n", cnt);
4     }
5     pthread_destroy();
6 }
7
8 void t2() {
9     for (int cnt = 0; cnt < 10; cnt++) {
10        printf("thread p2, loop: %d\n", cnt);
11    }
12    pthread_destroy();
13 }

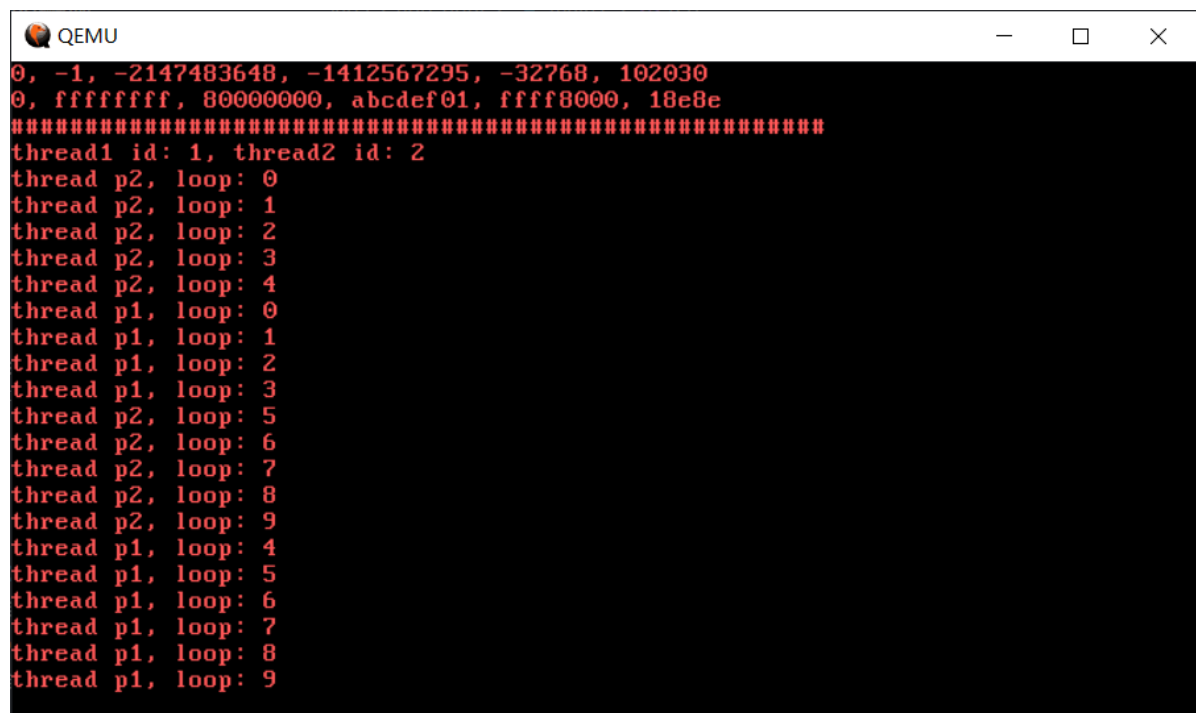
```

在主线程中创建这两个线程

```

1 uint32_t pid1 = pthread_create((uint32_t)t1);
2 uint32_t pid2 = pthread_create((uint32_t)t2);
3 printf("thread1 id: %d, thread2 id: %d\n", pid1, pid2);
4 pthread_destroy(); //destroy代替
5 return 0;

```



```

QEMU
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
#####
thread1 id: 1, thread2 id: 2
thread p2, loop: 0
thread p2, loop: 1
thread p2, loop: 2
thread p2, loop: 3
thread p2, loop: 4
thread p1, loop: 0
thread p1, loop: 1
thread p1, loop: 2
thread p1, loop: 3
thread p2, loop: 5
thread p2, loop: 6
thread p2, loop: 7
thread p2, loop: 8
thread p2, loop: 9
thread p1, loop: 4
thread p1, loop: 5
thread p1, loop: 6
thread p1, loop: 7
thread p1, loop: 8
thread p1, loop: 9

```

我们可以看到，p1, p2轮流执行，主线程先于这两个线程打印出了线程信息。

2. 访问全局变量

将上述测试样例中的cnt修改为全局变量，并在主线程中初始化为0。（该测试用例也是我们提交时的最终样例）

```

1 int cnt;
2
3 void t1() {
4     for (; cnt < 20; cnt++) {
5         printf("thread p1, loop: %d\n", cnt);
6     }

```

```

7   pthread_destroy();
8   }
9
10  void t2() {
11      for (; cnt < 10; cnt++) {
12          printf("thread p2, loop: %d\n", cnt);
13      }
14      pthread_destroy();
15  }

```

```

0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
*****
thread1 id: 1, thread2 id: 2
thread p1, loop: 0
thread p1, loop: 1
thread p1, loop: 2
thread p1, loop: 3
thread p1, loop: 4
thread p2, loop: 4
thread p2, loop: 5
thread p2, loop: 6
thread p2, loop: 7
thread p2, loop: 8
thread p2, loop: 9
thread p1, loop: 11
thread p1, loop: 12
thread p1, loop: 13
thread p1, loop: 14
thread p1, loop: 15
thread p1, loop: 16
thread p1, loop: 17
thread p1, loop: 18
thread p1, loop: 19

```

我们发现成功验证了不互斥地访问全局变量可能导致一些loop的缺失或重复，比如p1和p2都打印出loop 4，说明在执行写内存之前线程被切换出去。

该 challenge 修改的代码文件为 `memory.h`, `lib.h`, `irqHandle.c`, `kvm.c`, `syscall.c`, `app_print/main.c`, 并放在 `report/challenge/challenge3` 目录下, 完整代码需要在 `lab3` 需要 `git checkout k_thread` 切换到 `k_thread` 分支 (默认分支为 `exec` 分支)

• challenge 4

challenge4: (不写不扣分, 写了加分)

- 在 `create_thread` 函数里面, 我们要用线性时间搜索空闲 `tcb`, 你是否有更好的办法让它更快进行线程创建?
- 我们的这个线程库, 父子线程之间关联度太大, 有没有可能进行修改, 并实现一个调度器, 进行线程间自由切换? (这时 `tcb` 的结构可能需要更改, 一些函数功能也可以变化, 你可以说说思路, 也可以编码实现)
- 修改这个线程库或者自由设计一个线程库。

1. 维护一个空闲线程 `pid` 的栈, 初始时由于不存在线程因此该进程的线程栈中以次存放着所有线程号, 每当创建新线程时, 就将栈顶弹出索引到目标 `TDB` 表项, 当线程消亡时就将空出来的 `IP` 入栈
2. 可以采用 `round robin` 算法, 对于 `PCB` 中有 `ready` 的线程调入处理器执行, 无需记录父进程, 但是在 `create`, `yield`, `finish` 时仍需要准确维护 `esp` 和 `eip`, 需要注意的是要记录主线程为 0 号线程并且在其下的子线程完成之前。

• challenge 5

- challenge5: 你是否可以完善你的exec, 第三个参数接受变长参数, 用来模拟带有参数程序执行。
- 举个例子, 在shell里面输入cat a.txt b.txt, 实际上shell会fork出一个新的shell (假设称为shell0), 并执行exec("cat", "a.txt", "b.txt")运行cat程序, 覆盖shell0的进程。
- 不必完全参照Unix, 可以有自己的思考与设计。

实现了一个简单的文件查询, exec接受两个参数: 参数个数argc以及进程执行参数列表argv, 例如

```
1 char* argv[2] = {"app", "arg1"};
2 exec(2, argv); //exec(int argc, char** argv);
```

进入exec系统调用后会根据第一个参数的定位到待执行文件的文件名字符串, 并在文件列表中搜索存在的文件, 文件结构中保存了在磁盘上的位置以及扇区大小, 调用 loadelf 装载扇区内容。

3. 实现细节

• 3.1. timerHandle

timerHandle 负责每隔一段时间进行一次进程切换。具体而言, 当前进程会有四种状态:

- Running: 检查时间片是否用尽, 如果时间片用尽就由 RUNNABLE 的进程就直接切换到该进程执行即可, 如果没有就绪进程就将自己的时间计数再置为0直到中断到来并且有就绪进程能将当前进程换下。
- RUNNABLE: 当前进程不应该具有 RUNNABLE 状态, 所以这一情况会raise错误
- DEAD: 空白状态, 由 exit 系统调用将 RUNNING 状态设置成 DEAD 状态后主动调用 timerHandle
- BLOCKED: 由 sleep 系统调用将RUNNING状态设置成BLOCKED状态并设置睡眠时间后调用 timerHandle, 最后这两种状态要么检查有就绪态的进程执行, 要么就切换到 IDLE 进程等待有进程就绪为止。

• 3.2. syscallFork

选取一个空闲状态的 PCB 并创建一个子进程, 需要注意将父进程的地址空间和栈帧当中的内容 (包括PCB自身) 都 copy 到子进程的相应位置上(va_to_pa)

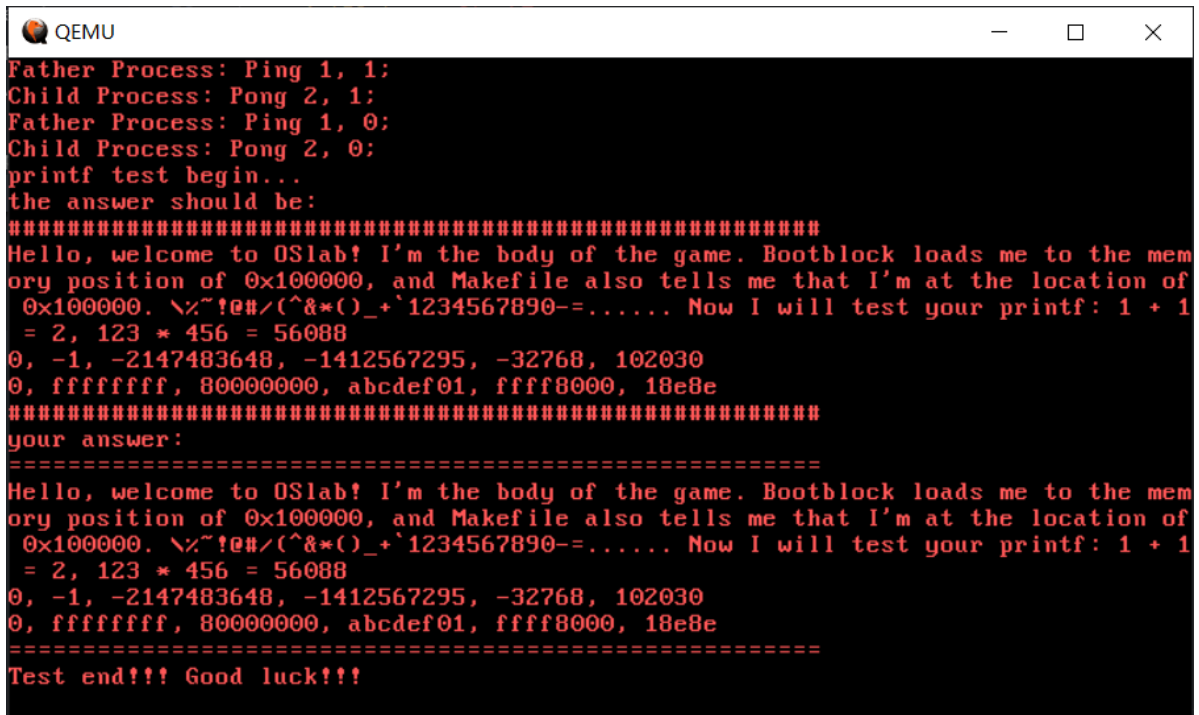
• 3.3.syscallExec

会将需要执行的可执行文件搬运到当前进程的地址空间并且重新初始化进程 (包括重要寄存器的初始化以及程序入口的指定等)

• 3.4. syscallSleep & syscallExit

没有需要额外注意的点, 不过可以再 syscallExit 处打印当前进程的信息便于 Debug

• 4. 实现成果



```
QEMU
Father Process: Ping 1, 1;
Child Process: Pong 2, 1;
Father Process: Ping 1, 0;
Child Process: Pong 2, 0;
printf test begin...
the answer should be:
#####
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the memory position of 0x100000, and Makefile also tells me that I'm at the location of 0x100000. \x~!@#/(^&*())_+'1234567890-=..... Now I will test your printf: 1 + 1 = 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
#####
your answer:
=====
Hello, welcome to OSlab! I'm the body of the game. Bootblock loads me to the memory position of 0x100000, and Makefile also tells me that I'm at the location of 0x100000. \x~!@#/(^&*())_+'1234567890-=..... Now I will test your printf: 1 + 1 = 2, 123 * 456 = 56088
0, -1, -2147483648, -1412567295, -32768, 102030
0, ffffffff, 80000000, abcdef01, ffff8000, 18e8e
=====
Test end!!! Good luck!!!
```

文件分布:

```
1 | lab3-201220154王紫萁
2 | | lab3
3 | | | lab3 //本次实验主要内容
4 | | | thread
5 | | | test //exercise时的小实验
6 | | | concurrent
7 | | | .git //默认处于exec分支下，challenge3的完整代码需要切换到k_thread
  | 分支下
8 | | | .gitignore
9 | | -
10 | | report
11 | | | lab3_report.pdf
12 | | | challenge
13 | | | | challenge3 //内核线程库相关
14 | | | - challenge5 //exec扩展相关
15 | | -
16 | -
```