LSCA开发日志

笔记本:	LSCA		
创建时间:	2020/6/28 10:31	**更新时间:**	2020/9/30 9:17
作者:	王恒宇		

2020年6月28日10:31:51
开始尝试编写demo，首先发现的一个问题是：程序中很多库函数并非直接就是glibc的系统调用（汇编级别），而是跳转到别的c库里再包装一层。例如许多c库函数最后都是调用的write或者read调用。这样就需要解决两个问题，第一个是将printf这种第一层的函数再alib中实现一遍；第二就是只拦截系统调用函数。

2020年7月14日07:29:34
反复查看项目需求后确定要取代glibc，因此需要将所有的库函数进行替换。今日准备完成共享内存任务执行机制的建立。目标是用户程序执行一条alibc定义的函数（printf），alibc记录log并将消息存放至共享内存。守护线程通过信号量同步共享内存中的数据，提取函数和参数并执行，将执行结果返回至指定共享内存并按照ATPCS约定返回参数。

使用信号量解决了互斥问题，但是由于没有记录型信号量所以存在忙等。另外对于共享内存缓冲区的设计采用环形队列。想一天就做好上面的事情着实有点异想天开了。

2020年7月15日11:09:59
经过查找确定了两个问题。1.多线程环境下存在记录型信号量的库，线程可以等待某条件而阻塞。但是进程没有这么方便。2.进程间信号量库没有记录型，消息机制或许可以更好解决问题，但是猜测消息开销较大。目前
决定先使用自旋的信号量，低成本的消息机制以后再探索。

2020年7月18日16:54:10
完成绘制lsca角色关系及流程图，下一步设计任务和结果结构体，并构造相应的缓冲区

```
5 [File Path]: /home/euler/openEuler.git/kernel/include/uapi/asm-generic/unistd.h [Ex Command]: 1
```

系统调用位置。

2020年7月24日23:38:25

编写一个传参的函数，可以发现参数传递时候倒序存数据，用栈存8个以外的参数，剩下的存在0-7寄存器中

```c
#include <stdio.h>
void func(int a,int b,int c,int d,int e,int f,int g,int h,int i,int j);
void func(int a,int b,int c,int d,int e,int f,int g,int h,int i,int j){

        printf("%x,%x,%x,%x,%x,%x,%x,%x,%x,%x",a,b,c,d,e,f,g,h,i,j);
}

int main()
{
        func(1,2,3,4,5,6,7,8,9,10);
        return 0;
}
```

```
0000000000000724 <func>:
 724:   d10143ff        sub     sp, sp, #0x50
 728:   a9027bfd        stp     x29, x30, [sp, #32]
 72c:   910083fd        add     x29, sp, #0x20
 730:   b9002fa0        str     w0, [x29, #44]
 734:   b9002ba1        str     w1, [x29, #40]
 738:   b90027a2        str     w2, [x29, #36]
 73c:   b90023a3        str     w3, [x29, #32]
 740:   b9001fa4        str     w4, [x29, #28]
 744:   b9001ba5        str     w5, [x29, #24]
 748:   b90017a6        str     w6, [x29, #20]
 74c:   b90013a7        str     w7, [x29, #16]
 750:   90000000        adrp    x0, 0 <_init-0x598>
 754:   91226008        add     x8, x0, #0x898
 758:   b9403ba0        ldr     w0, [x29, #56]
 75c:   b90013e0        str     w0, [sp, #16]
 760:   b94033a0        ldr     w0, [x29, #48]
 764:   b9000be0        str     w0, [sp, #8]
 768:   b94013a0        ldr     w0, [x29, #16]
 76c:   b90003e0        str     w0, [sp]
 770:   b94017a7        ldr     w7, [x29, #20]
 774:   b9401ba6        ldr     w6, [x29, #24]
 778:   b9401fa5        ldr     w5, [x29, #28]
 77c:   b94023a4        ldr     w4, [x29, #32]
 780:   b94027a3        ldr     w3, [x29, #36]
 784:   b9402ba2        ldr     w2, [x29, #40]
 788:   b9402fa1        ldr     w1, [x29, #44]
 78c:   aa0803e0        mov     x0, x8
 790:   97ffffa0        bl      610 <printf@plt>
 794:   d503201f        nop
 798:   a9427bfd        ldp     x29, x30, [sp, #32]
 79c:   910143ff        add     sp, sp, #0x50
 7a0:   d65f03c0        ret
```

```
00000000000007a4 <main>:
 7a4:   d10083ff        sub     sp, sp, #0x20
 7a8:   a9017bfd        stp     x29, x30, [sp, #16]
 7ac:   910043fd        add     x29, sp, #0x10
 7b0:   52800140        mov     w0, #0xa                        // #10
 7b4:   b9000be0        str     w0, [sp, #8]
 7b8:   52800120        mov     w0, #0x9                        // #9
 7bc:   b90003e0        str     w0, [sp]
 7c0:   52800107        mov     w7, #0x8                        // #8
 7c4:   528000e6        mov     w6, #0x7                        // #7
 7c8:   528000c5        mov     w5, #0x6                        // #6
 7cc:   528000a4        mov     w4, #0x5                        // #5
 7d0:   52800083        mov     w3, #0x4                        // #4
 7d4:   52800062        mov     w2, #0x3                        // #3
 7d8:   52800041        mov     w1, #0x2                        // #2
 7dc:   52800020        mov     w0, #0x1                        // #1
 7e0:   97ffffd1        bl      724 <func>
 7e4:   52800000        mov     w0, #0x0                        // #0
```

```
7e8:   a9417bfd        ldp     x29, x30, [sp, #16]
7ec:   910083ff        add     sp, sp, #0x20
7f0:   d65f03c0        ret
7f4:   00000000        .inst   0x00000000 ; undefined
```

通过观察静态汇编函数发现代码段中有多种系统调用，也看到了用于write的64号系统调用，对于的glibc函数名是__libc__write.整个过程中64号系统调用被执行了三次并且三次都是在一起的。所以真的不知道glibc库封装之后使用了些什么系统调用完成功能。

2020年7月26日23:35:49

完成了echo_server 和echo_client 涉及到的系统调用分别如下。

```
root@lsca:~/pro/test/test_for_echo# cat st
execve("./server", ["./server"], 0xffffff3742d0 /* 24 vars */) = 0
brk(NULL)                               = 0xaaaadef31000
faccessat(AT_FDCWD, "/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=34135, ...}) = 0
mmap(NULL, 34135, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffff86874000
close(3)                                = 0
faccessat(AT_FDCWD, "/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0 \10\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1345176, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffff86872000
mmap(NULL, 1413976, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xffff866f9000
mprotect(0xffff8683a000, 61440, PROT_NONE) = 0
mmap(0xffff86849000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x140000) = 0xffff86849000
mmap(0xffff8684f000, 13144, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xffff8684f000
close(3)                                = 0
mprotect(0xffff86849000, 16384, PROT_READ) = 0
mprotect(0xaaaadbb76000, 4096, PROT_READ) = 0
mprotect(0xffff8687f000, 4096, PROT_READ) = 0
munmap(0xffff86874000, 34135)           = 0
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
bind(3, {sa_family=AF_INET, sin_port=htons(12345), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
listen(3, 10)                           = 0
accept(3, NULL, NULL)                   = 4
close(3)                                = 0
read(4, "xixi\n", 1000)                 = 5
write(4, "xixi\n", 5)                   = 5
read(4, "jaja\n", 1000)                 = 5
write(4, "jaja\n", 5)                   = 5
read(4, "haha\n", 1000)                 = 5
write(4, "haha\n", 5)                   = 5
read(4, "", 1000)                       = 0
close(4)                                = 0
accept(3, NULL, NULL)                   = -1 EBADF (Bad file descriptor)
exit_group(0)                           = ?
+++ exited with 0 +++
```

```
root@lsca:~/pro/test/test_for_echo# cat ct
execve("./client", ["./client", "localhsot"], 0xffffcf217208 /* 24 vars */) = 0
brk(NULL)                               = 0xaaab0adfb000
faccessat(AT_FDCWD, "/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=34135, ...}) = 0
mmap(NULL, 34135, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffffa2555000
close(3)                                = 0
faccessat(AT_FDCWD, "/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0 \10\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1345176, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffa2553000
mmap(NULL, 1413976, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xffffa23da000
mprotect(0xffffa251b000, 61440, PROT_NONE) = 0
mmap(0xffffa252a000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x140000) = 0xffffa252a000
mmap(0xffffa2530000, 13144, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xffffa2530000
close(3)                                = 0
mprotect(0xffffa252a000, 16384, PROT_READ) = 0
mprotect(0xaaaad2bf0000, 4096, PROT_READ) = 0
mprotect(0xffffa2560000, 4096, PROT_READ) = 0
munmap(0xffffa2555000, 34135)           = 0
socket(AF_INET, SOCK_STREAM, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(12345), sin_addr=inet_addr("0.0.0.0")}, 16) = 0
fstat(0, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 1), ...}) = 0
brk(NULL)                               = 0xaaab0adfb000
brk(0xaaab0ae1c000)                     = 0xaaab0ae1c000
read(0, "xixi\n", 1024)                 = 5
write(3, "xixi\n", 5)                   = 5
read(3, "xixi\n", 1000)                 = 5
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 1), ...}) = 0
write(1, "xixi\n", 5)                   = 5
read(0, "jaja\n", 1024)                 = 5
write(3, "jaja\n", 5)                   = 5
read(3, "jaja\n", 1000)                 = 5
write(1, "jaja\n", 5)                   = 5
read(0, "haha\n", 1024)                 = 5
write(3, "haha\n", 5)                   = 5
read(3, "haha\n", 1000)                 = 5
write(1, "haha\n", 5)                   = 5
read(0, 0xaaab0adfb260, 1024)           = ? ERESTARTSYS (To be restarted if SA_RESTART is set)
```

其中涉及到的不只是socket系列以及read和write，还有很多初始化程序所需要的mmap这类系统调用。目前的计划是先排除所有程序都要用的部分（也就是faccessat、mmap这类）先做最强相关的socket、write、read。

```
/* fs/read_write.c */
#define __NR3264_lseek 62
__SC_3264(__NR3264_lseek, sys_llseek, sys_lseek)
#define __NR_read 63
__SYSCALL(__NR_read, sys_read)
#define __NR_write 64
__SYSCALL(__NR_write, sys_write)
#define __NR_readv 65
__SC_COMP(__NR_readv, sys_readv, compat_sys_readv)
#define __NR_writev 66
__SC_COMP(__NR_writev, sys_writev, compat_sys_writev)
#define __NR_pread64 67
__SC_COMP(__NR_pread64, sys_pread64,
compat_sys_pread64)
#define __NR_pwrite64 68
__SC_COMP(__NR_pwrite64, sys_pwrite64,
compat_sys_pwrite64)
#define __NR_preadv 69
__SC_COMP(__NR_preadv, sys_preadv, compat_sys_preadv)
#define __NR_pwritev 70
```

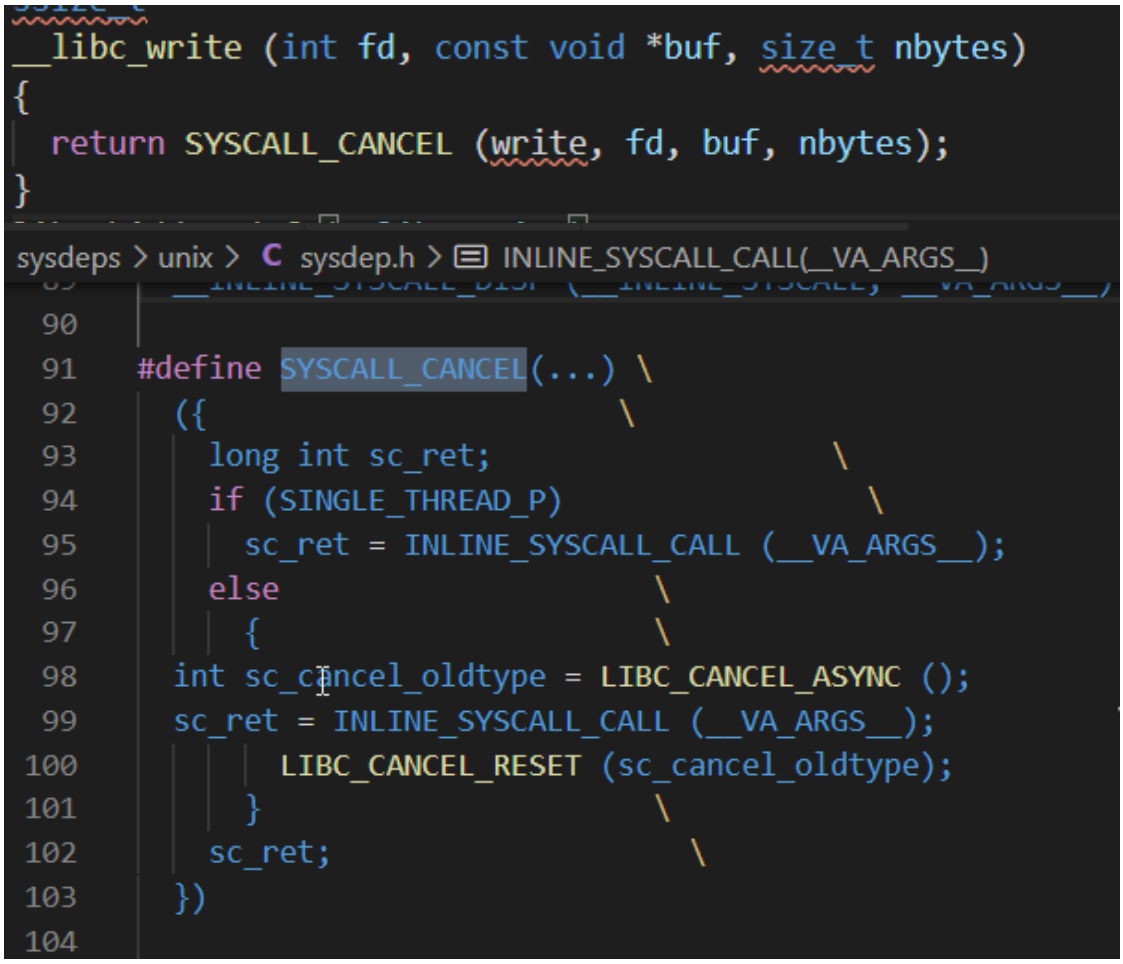
__SC_COMP(__NR_pwritev, sys_pwritev, compat_sys_pwritev)

```
  /* net/socket.c */
#define __NR_socket 198
__SYSCALL(__NR_socket, sys_socket)
#define __NR_socketpair 199
__SYSCALL(__NR_socketpair, sys_socketpair)
#define __NR_bind 200
__SYSCALL(__NR_bind, sys_bind)
#define __NR_listen 201
__SYSCALL(__NR_listen, sys_listen)
#define __NR_accept 202
__SYSCALL(__NR_accept, sys_accept)
#define __NR_connect 203
__SYSCALL(__NR_connect, sys_connect)
#define __NR_getsockname 204
__SYSCALL(__NR_getsockname, sys_getsockname)
#define __NR_getpeername 205
__SYSCALL(__NR_getpeername, sys_getpeername)
#define __NR_sendto 206
__SYSCALL(__NR_sendto, sys_sendto)
#define __NR_recvfrom 207
__SC_COMP(__NR_recvfrom, sys_recvfrom,
compat_sys_recvfrom)
#define __NR_setsockopt 208
__SC_COMP(__NR_setsockopt, sys_setsockopt,
compat_sys_setsockopt)
#define __NR_getsockopt 209
__SC_COMP(__NR_getsockopt, sys_getsockopt,
compat_sys_getsockopt)
#define __NR_shutdown 210
__SYSCALL(__NR_shutdown, sys_shutdown)
#define __NR_sendmsg 211
__SC_COMP(__NR_sendmsg, sys_sendmsg,
compat_sys_sendmsg)
#define __NR_recvmsg 212
__SC_COMP(__NR_recvmsg, sys_recvmsg,
compat_sys_recvmsg)
```

2020年7月27日14:28:26

研究glibc对于write的封装。一开始在glibc的io文件夹下找到了write.c代码，发现就是一些判断和弱别名的声明，没有什么实际作

用的代码。后来发现在sysdeps/unix/sysv\linux下，并且看了read、write和socket之后发现都是一个模子套出来的，就是跳转到宏SYSCALL_CANCEL。找到其定义在sysdep.h。



```
__libc_write (int fd, const void *buf, size_t nbytes)
{
  return SYSCALL_CANCEL (write, fd, buf, nbytes);
}
```

sysdeps > unix > C sysdep.h > INLINE_SYSCALL_CALL(__VA_ARGS__)

```
90
91  #define SYSCALL_CANCEL(...) \
92    ({                        \
93      long int sc_ret;                  \
94      if (SINGLE_THREAD_P)                \
95        sc_ret = INLINE_SYSCALL_CALL (__VA_ARGS__);
96      else                      \
97        {                       \
98    int sc_cancel_oldtype = LIBC_CANCEL_ASYNC ();
99    sc_ret = INLINE_SYSCALL_CALL (__VA_ARGS__);
100           LIBC_CANCEL_RESET (sc_cancel_oldtype);
101       }                       \
102     sc_ret;                     \
103   })
104
```

其主要部分又是对于不同参数数量的适配。发现一篇讲arm系统调用的不错的文章。

https://www.cnblogs.com/Five100Miles/p/8878080.html

```
#define __INLINE_SYSCALL0(name) \
  INLINE_SYSCALL (name, 0)
#define __INLINE_SYSCALL1(name, a1) \
  INLINE_SYSCALL (name, 1, a1)
#define __INLINE_SYSCALL2(name, a1, a2) \
  INLINE_SYSCALL (name, 2, a1, a2)
#define __INLINE_SYSCALL3(name, a1, a2, a3) \
  INLINE_SYSCALL (name, 3, a1, a2, a3)
#define __INLINE_SYSCALL4(name, a1, a2, a3, a4) \
  INLINE_SYSCALL (name, 4, a1, a2, a3, a4)
#define __INLINE_SYSCALL5(name, a1, a2, a3, a4, a5) \
  INLINE_SYSCALL (name, 5, a1, a2, a3, a4, a5)
#define __INLINE_SYSCALL6(name, a1, a2, a3, a4, a5, a6) \
  INLINE_SYSCALL (name, 6, a1, a2, a3, a4, a5, a6)
#define __INLINE_SYSCALL7(name, a1, a2, a3, a4, a5, a6, a7) \
  INLINE_SYSCALL (name, 7, a1, a2, a3, a4, a5, a6, a7)
```

找到最终INLINE-SYSCALL的定义，但是发现这个name无从知晓。结合看到的make-syscall.sh和template.S猜测是用到这部分代码的时候自动生成的，不是源代码里能看到的。大佬们在编写系统的时候用了太多奇技淫巧了，我发现我真的对c语言和系统编程一无所知。。。

```
INLINE_SYSCALL(name, nr, args...) __syscall_##name (args
```

从这里也可以看出哪些write.S之类的用脚本生成的文件早就不存在于glibc里了，这也是为什么写openEuler的书的时候找不到具体系统调用的汇编代码了。但是静态编译再反汇编之后可以看到。

```
Replaced all simple system call files *.S throughout sysdeps/unix
with syscalls.list files to be processed by make-syscalls.sh.
```

在这个过程中了解到glibc并不是专为linux开发的，而是要考虑多个操作系统，很多调用的名字用宏定义改改去，目前了解到的原因之一就是：兼容不同芯片类型，如64位和32位的差别，将传参的long统一为int等。

最近两天看了很多网上的帖子还有书，但是对于这个复杂的库和系统的联系还是感到一知半解没有一个完全明确的了解。但是书的质量还是高一些的，情景分析这本就不错，但是他讲系统调用主要集中在内核部分，而我想要寻找的是glibc部分，他用反汇编巧妙的跳过了glibc部分。

2020年8月3日12:06:49

再次确定项目整体设计。首先，我们的目标是使得lsca代替glibc甚至内核，使得用户程序不需要陷入内核就可以完成系统调用。一个用户程序所使用的glibc库函数部分是在用户态下就可以完成的，部分则需要使用到敏感指令，需要内核协助。因此，lsca若想要代替内核就需要在两个特权级上都部署服务，分别处理普通指令即可完成的系统调用以及涉及敏感指令的系统调用。接下来的计划总体上是首先完成用户态的任务，再完成内核态的任务。过程遇到的需要内核态完成的系统调用先由用户态daemon协助完成。

接下来计划完成的任务是：1.完成链接库设计2.完成用户态daemon设计3.内核系统调用替换调试。

2020年8月7日00:22:34
换到了clion开发，参数传递目前问题不大，采用的方案是将字数变为字符串在变为指定变量，字符串指针采用strcpy复制到缓冲区内传送，解决栈隔离问题。目前节点设计和实现测试通过。下一步完成环形队列的设计实现，再进行联调。

2020年8月8日01:58:31
环形队列测试完成，但内存申请还是malloc而不是共享内存，需要替换一下。今天效率较低，其实总工作的时间只有一两个小时吧。下面完善链接库关于write和read的调用封装，与daemon联调。

2020年8月9日22:27:33
完成封装对于共享内存和信号量的各种操作。遇到的新问题是模块化，也就是能像linux kernel一样仅仅修改两个头文件就能够方便地加入新的系统调用。搜索了一番之后还没有找到特别方便的方法，也不了解Linux kernel的方法。但是目前不打算在此纠结，目前的想法还是先能用再说，模块化、速度优化等等往后放一放，因为可能还没优化或者还没模块化就发现有些路子行不通。

2020年8月10日01:11:53
daemon完成系统调用代理逻辑时应该释放信号量，否则处理时间太长可能出现错误，只在读写的时候占用信号量。因此在daemon处理task时候应该放开信号量。整体流程大致走完了。还需要完善agent_write封装。尝试了函数指针数组、tcc即时编译非常有趣。

2020年8月10日18:54:07
替换方法 LD_PRELOAD=./agent_lib.so ./user
使用共享内存遇到的一个情况：当写进程向指针指向的位置写入后，读进程读取会导致段错误。也就是读进程访问到了不该访问的内存。如果共享内存中声明的是数组或非指针的变量就不会有问题。而我使用的数据结构涉及到指针的嵌套，而由于对指针缺乏深刻的理解。目前合理的猜想是因为在写数据的时候看起来像是向指针指向的地方写入了数据，但实际上是将共享的指针指向了当前进程的私有空间，导致另一个进程在访问是遇到段错误。

2020年8月13日12:31:58
两天前就已经完成了demo的工作。
当下要做的事情是:1.编写具有自适应能力的makefile。2.模块化wrapper以及相应的syscall_agents。3.编写中期报告。

1.makefile
首先是在各个子文件夹下建立makefile，将每个.c文件根据头文件得出依赖关系，进而编译成为.o。
第二步编译顶层的daemon和lib的makefile，借助子makefile完成目标文件daemon和lib.so的生成。

2020年8月21日22:47:06
假期结束，继续开始开发，15日凌晨完成了中期报告。其中t1文件夹下的remote是Clion的远程文件夹，lsca是上传到github的文件夹。现在两者是一样的版本，但是在lsca中多了video文件夹和中期文档，计划将remote文件夹弃用，改用lsca文件夹，并且以后在此文件夹中维护版本，将其他文件夹存档，减少文件夹数量。

当前kunpeng-ubuntu root目录下lsca为版本维护文件夹，tcc为实验tcc的文件夹，以前的root目录下的文件转移到backup文件夹中。

2020年9月8日16:44:29
恢复LSCA开发，下午看了一下序列化的方法，都是封装结构体。但目前存在的问题是用于存储参数的结构体有很多空余的位置可能会被放入内存占地方。所以决定先不管序列化的事情，先解决echo用

到的多种调用，以及将wrapper文件分离问题。c语言输入输出重定向方法剪藏在印象笔记了。

2020年9月20日21:30:58
read和write这类在终端进行输出输出的操作理论上是需要输入和输出重定向的，比较快速的解决方案是将需要输入输出的程序与daemon运行在同一终端中。目前测试两个daemon同时在不同的终端运行时可以的。后续
有时间可以改重定向，但是需要在传递参数的时候加入描述终端的信息。

2020年9月23日21:17:37
在实现socket wrapper时候发现端口和文件描述符需要进行绑定。但是，同一个文件描述符即一个数字，对应在不同的进程里对应的打开的文件是不同的。也就是每个进程中都有独立的文件描述符系统，当进程打开一个文件时由内核分配。

2020年9月25日13:46:41
之前尝试将内存调大之后会core dump，但是今天遇到的有趣的事情是，调小了也会，所以我怀疑是因为程序没有正常结束，共享内存没有被正常释放，导致下次程序再次使用同一片共享内存时发生了错误。重启后尝试了调大的共享区，可以正常使用。

2020年9月28日22:09:39
最近两天基本上是在解决各种莫名其妙的问题，明明是共用的共享内存，但是daemon在agentLread的时候就会发生段错误然后就核心转储了。。。调试了半天发现，原来是用于测试的打印字符串，使用了%s的占位符，但是后面用的变量却是int，感觉是被越界访问了，write程序没有找到\0，就一直往后读，就越界了，于是触发段错误。

为什么一开始会认为是共享内存的问题呢，因为之前发生这个错误就是共享内存指针的指针搞错了位置，导致越界，这次就一直盯着memset和memcpy看来看去，废了半天劲，发现是用于测试的不起眼的句子出了问题，这个。。以后出问题还是一句一句看，别觉得有的东西不重要就放掉了，导致浪费了大半天时间。

2020年9月29日22:19:18

项目要求的测试样例echo分别测试client和server在kunpeng服务器的openEuler上已经通过。

值得优化的地方是输入输出的重定向，但是试了一堆方法目前还没有好用的。。。对于server程序是没有影响的。但是对于需要输入的程序，例如client或者其他和daemon不运行在一个终端上的程序，daemon默认去自己的stdin下读数据，但是输入却是在被代理的程序的终端，因此会导致daemon空等。凑合的办法是在daemon的终端输入，但是不够体面，后面再探索一下重定向方法。

2020年9月30日08:47:46
总结了一下后续需要改进和优化的地方，作为后期的目标。
1.多线程。在daemon接收到请求后，会根据请求的系统调用号调用函数数组中对应的代理函数，但是这个过程是单线程的，因此如果有一个像echo程序server端一样需要阻塞、等待的程序调用了daemon的服务，则会导致daemon无法为其他函数提供服务。因此需要在进入代理函数数组之前开辟新的线程或进程代理执行，方便其他进程请求daemon服务。

2.缓冲区及序列化器。当前的缓冲区和序列化部分是一个简单的、方便上手的设计，但是很不精致，对于新加系统调用代理来说是不够友好。使用的便捷性方面，一个理想的序列化器应该是能够识别参数数量和类型，一个函数打包好所有需求。性能方面，由于系统调用很常用，因此打包过程重复次数很多，需要尽可能精简打包过程，保障速度。空间占用方面，尽量保障缓冲区是弹性的，随着代理系统调用的量对大小进行增减，可以牺牲部分空间占用换取性能，不用一直调整，但整体上看起来应该是均衡的。

3.活动日志。目前的活动日志更多只是一个示意，直接写入到文件中的。但是对于系统调用这种高频操作，文件读写越少越好。我想或许应该先写入到一片内存缓冲区内，等待系统调用代理任务不繁忙时再持久化到硬盘。当然这个方案如果是在本地单机上运行可能还需要考虑突然断电和设备损坏的影响，但是云上的虚拟化环境中应该比较有保障。另外我觉得应该有对应这种需求的中间件存在了，后期调研一下。

4.重定向。值得优化的地方是输入输出的重定向，试了一堆方法目前还没有好用的。对于类似echo_server程序是没有影响的。但是对于需要输入的程序，例如client或者其他和daemon不运行在一个终端上的程序，daemon默认去自己的stdin下读数据，但是输入却是在被代理的程序的终端，因此会导致daemon空等。凑合的办法是在daemon的终端输入，但是不够体面，后面再探索一下重定向方法。

以上就是目前能想到的需要优化的地方，后续随项目进展继续补充。