



清华大学
Tsinghua University

全国操作系统能力大赛(CSCC 2025)工作总结

宋建昊 Undefined-OS

2025年09月23日

比赛流程与评分标准

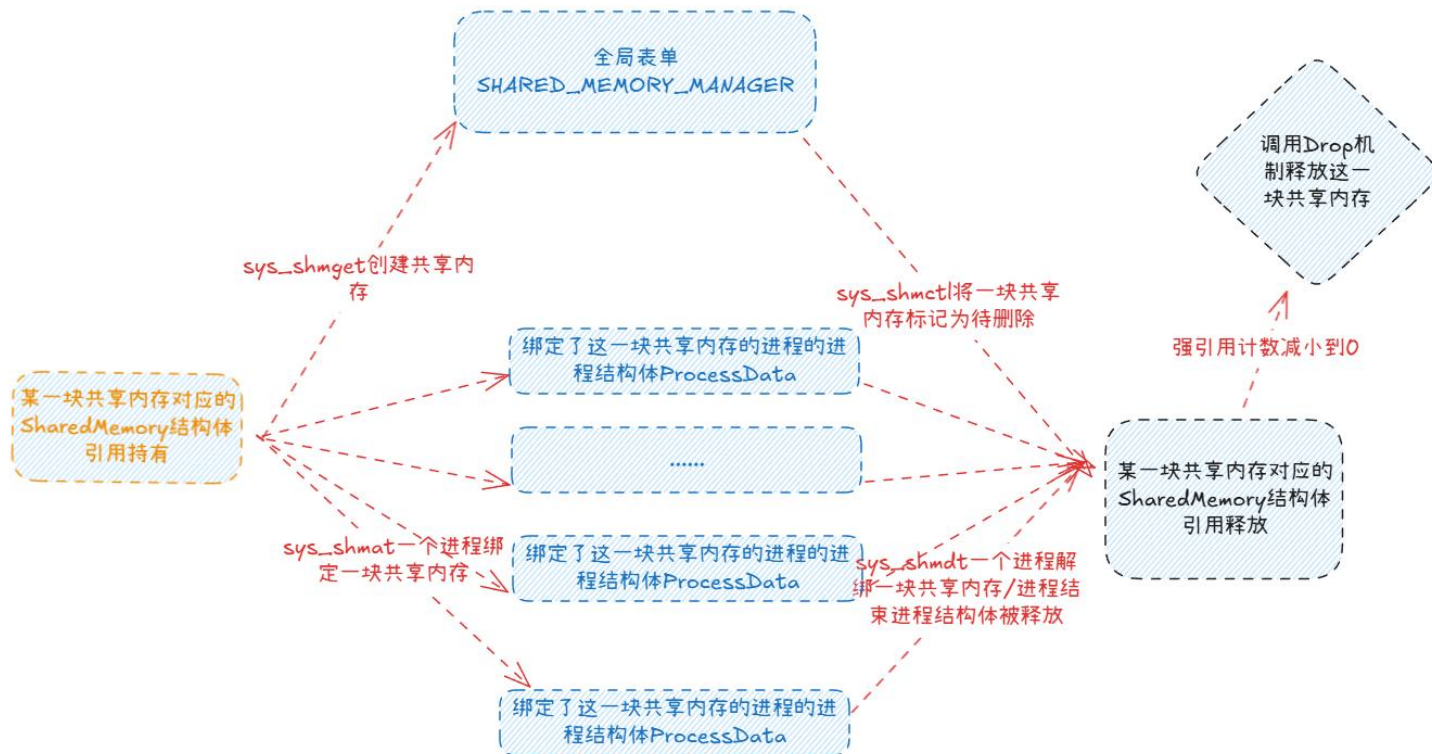
- 初赛：初赛测例+文档
- 决赛：初赛测例+决赛新增测例+现场赛测例+文档+答辩
- 部分初赛测例：basic/busybox/iozone/libcbench/lmbench/ltp/lua
- 部分现场赛测例：git(本地+网络)/vim/gcc/rustc(思路是“OS自举”，即在自己的OS上能够完成OS开发的所有流程)
- 架构：Riscv/Loongarch
- C标准库实现：glibc/musl

我参与解决的问题：iozone测例（共享内存读写+性能）

- 我参与实现了Undefined-OS的共享内存模块，实现了(sys_shmget/shmat/sys_shmctl/sys_shmdt)等几个共享内存相关的系统调用。
- 设计中利用了Rust语言的智能指针(Arc)和自动Drop机制来优雅地实现共享内存的分配/映射/释放。
- 最终Undefined-OS成功通过了iozone测试。

共享内存实现简介

```
pub struct SharedMemory {  
    /// The key of the shared memory segment  
    pub key: u32,  
    /// Virtual kernel address of the shared memory segment  
    pub addr: usize,  
    /// Page count of the shared memory segment  
    pub page_count: usize,  
}  
  
impl Drop for SharedMemory {  
    fn drop(&mut self) {  
        let allocator = global_allocator();  
        allocator.dealloc_pages(self.addr, self.page_count);  
    }  
}  
  
pub struct SharedMemoryManager {  
    mem_map: Mutex<BTreeMap<u32, Arc<SharedMemory>>>,  
    next_key: AtomicU32,  
}  
  
impl SharedMemoryManager {  
    //...  
    pub fn delete(&self, key: u32) -> bool {  
        // sys_shmctl在IPC_RMID参数下的行为就是将对应的SharedMemory从全局的SHARED_MEM  
        self.mem_map.lock().remove(&key).is_some()  
    }  
}  
  
pub static SHARED_MEMORY_MANAGER: SharedMemoryManager = SharedMemoryManager::new  
  
pub struct ProcessData {  
    //方便sys_shmctl找到对应的SharedMemory, 其参数是一个虚拟地址  
    /// Shared memory  
    pub shared_memory: Mutex<BTreeMap<VirtAddr, Arc<SharedMemory>>>,  
}
```



- 答辩过程中评审老师的提问：共享内存实现过程中如何避免循环引用？
- 进程结构体持有共享内存块的引用，但是共享内存块不持有进程结构体的引用。
- 智能指针的Drop机制使得内存回收自动进行，不存在通过共享内存块遍历/找到绑定它的进程这种情景，所以共享内存块可以不持有进程结构体的引用

一些设计细节

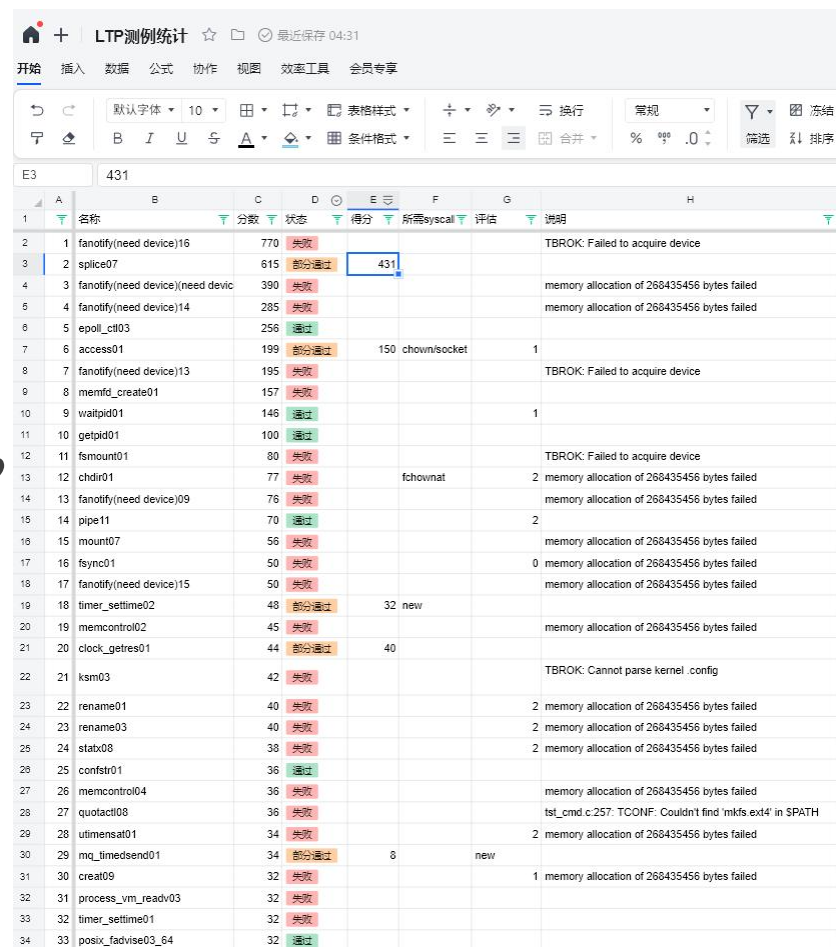
- 在sys_shmctl将一个共享内存块设置为待删除时，我们的做法是直接将其从全局表单中移除，被sys_shmctl标记为待删除的共享内存段不应还能够被sys_shmat映射，我们直接删除的做法可以让全局表单中查不到这个共享内存段，从源头杜绝了这种行为，否则还需要再sys_shmat中进行判定校验。
- ProcessData的shared_memory类型定义为Mutex<BTreeMap<VirtAddr, Arc<SharedMemory>>>，原因是sys_shmdt的参数类型是shm_addr，这样设置方便直接找到虚拟地址所对应的共享内存段，每一个类型和参数的设计都有其背后的原因。

功能性(垃圾回收)和标准的trade-off

- 关于共享内存自动释放的问题，有时会面临一种特殊的情况——一段通过共享内存段，如果所有通过`sys_shmat`绑定这段内存的进程都通过`sys_shmdt`解绑了，但是没有任何进程调用`sys_shmctl`把它标记为待删除，操作系统是否应该释放这段内存？此时一般有两种做法。
- 全局表单中SharedMemoryManager中使用强引用，也就是我们现在的实现方式，只有在有进程调用了`sys_shmctl`把某段共享内存设置为待删除同时所有绑定的进程解绑的时候才会进行释放，充分尊重了用户程序的行为，但是在用户程序不规范的情况下会造成内存泄露。
- 全局表单中SharedMemoryManager中使用弱引用，即使没有进程调用了`sys_shmctl`把某段共享内存设置为待删除，只要所有绑定的进程解绑就进行释放，一定程度上解决了内存释放的问题，但是OS无法假设所有进程解绑后共享内存就不再需要，因为其他进程可能在未来重新通过`sys_shmat`附加到该内存段
- 面临这种trade-off，我们去查阅了linux操作系统的行为，具体如下：
- 通过`sys_shmget`创建的共享内存段是持久性系统级资源，声明周期独立于创建它的进程。它的生命周期不与任何单个进程绑定。即使所有进程都已调用`sys_shmdt`解绑或进程终止，共享内存段仍然存在于内核中，占用内存资源，直到显式调用`sys_shmctl`或系统重启，总体来说采取前者的实现，尊重了标准牺牲了垃圾回收的功能。

体力活：ltp测例（系统调用的鲁棒性和完备性）

- ltp(linux test project)测例是一个数量庞大的完备的测例集合，会系统地测试每一个系统调用在各种情况下(正常返回，各种错误情况)的行为和返回值是否符合预期。ltp测例的总分数远高于其他所有测例之和，ltp分数基本决定了队伍测例得分的水平 and 排名。
- 每个ltp测例的分数不同，有的高达几百分，有的低至1分，我手动通过人工和编写脚本的方式测试了ltp的几乎所有测例，对于测例的分值和实现难度进行评估，以便制定在有限时间内最高效的实现顺序和得分策略。
- 基于评估结果的顺序，我修复和完善了大量syscall的行为和错误处理，使其能够通过ltp测试。
- 最终Undefined-OS的ltp测例分数在初赛中排名第二。



The screenshot shows a spreadsheet titled "LTP测例统计" (LTP Test Results Summary). It contains a table with columns for test name, score, status, and description. The table lists various system call tests and their results.

| 名称 | 分数 | 状态 | 得分 | 所需syscall | 评估 | 说明 |
|------------------------------------|-----|------|-----|--------------|----|--|
| fanotify(need device)16 | 770 | 失败 | | | | TBROK: Failed to acquire device |
| splice07 | 615 | 部分通过 | 431 | | | |
| fanotify(need device)(need device) | 390 | 失败 | | | | memory allocation of 268435456 bytes failed |
| fanotify(need device)14 | 285 | 失败 | | | | memory allocation of 268435456 bytes failed |
| epoll_ctl03 | 256 | 通过 | | | | |
| access01 | 199 | 部分通过 | 150 | chown/socket | 1 | |
| fanotify(need device)13 | 195 | 失败 | | | | TBROK: Failed to acquire device |
| memfd_create01 | 157 | 失败 | | | | |
| waitpid01 | 146 | 通过 | | | | 1 |
| getpid01 | 100 | 通过 | | | | |
| fsmount01 | 80 | 失败 | | | | TBROK: Failed to acquire device |
| chdir01 | 77 | 失败 | | fchownat | 2 | memory allocation of 268435456 bytes failed |
| fanotify(need device)09 | 76 | 失败 | | | | memory allocation of 268435456 bytes failed |
| pipe11 | 70 | 通过 | | | | 2 |
| mount07 | 56 | 失败 | | | | memory allocation of 268435456 bytes failed |
| fsync01 | 50 | 失败 | | | | 0 memory allocation of 268435456 bytes failed |
| fanotify(need device)15 | 50 | 失败 | | | | memory allocation of 268435456 bytes failed |
| timer_settime02 | 48 | 部分通过 | 32 | new | | |
| memcontrol02 | 45 | 失败 | | | | memory allocation of 268435456 bytes failed |
| clock_getres01 | 44 | 部分通过 | 40 | | | |
| ksm03 | 42 | 失败 | | | | TBROK: Cannot parse kernel config |
| rename01 | 40 | 失败 | | | | 2 memory allocation of 268435456 bytes failed |
| rename03 | 40 | 失败 | | | | 2 memory allocation of 268435456 bytes failed |
| statx08 | 38 | 失败 | | | | 2 memory allocation of 268435456 bytes failed |
| conftst01 | 36 | 通过 | | | | |
| memcontrol04 | 36 | 失败 | | | | memory allocation of 268435456 bytes failed |
| quotactl08 | 36 | 失败 | | | | test_cmd c:257: TCONF: Couldn't find 'mkfs_ext4' in \$PATH |
| utimensat01 | 34 | 失败 | | | | 2 memory allocation of 268435456 bytes failed |
| mq_timedsend01 | 34 | 部分通过 | 8 | new | | |
| creat09 | 32 | 失败 | | | | 1 memory allocation of 268435456 bytes failed |
| process_vm_readv03 | 32 | 失败 | | | | |
| timer_settime01 | 32 | 失败 | | | | |
| posix_fadvise03_64 | 32 | 通过 | | | | |

例子：dup3/setpgid/getpgid/shmget

```
pub fn sys_setpgid(pid: u32, pgid: u32) -> LinuxResult<isize> {
    let process = if pid == 0 {
        current_process()
    } else {
        if pid == current_process().get_pid(){
            current_process()
        } else {
            current_process().get_child(pid).ok_or(LinuxError::ESRCH)?.clone()
        }
    };
    if pgid == 0 {
        process.create_group();
    } else if pgid < 0 || pgid > 4194304 {
        return Err(LinuxError::EINVAL);
    } else {
        if !process.move_to_group(pgid) {
            return Err(LinuxError::EPERM);
        }
    }
    Ok(0)
}

pub fn sys_getpgid(pid: u32) -> LinuxResult<isize> {
    let process = if pid == 0 {
        current_process()
    } else {
        get_process(pid).ok_or(LinuxError::ESRCH)
    };
    Ok(process.get_group().get_pgid() as _)
}
```

```
pub fn sys_shmget(key: c_int, size: c_ulong, shm_flag: c_int) -> LinuxResult<isize> {
    let size = size as usize;
    let flags = ShmFlags::from_bits_truncate(shm_flag);
    // TODO: permission check
    if key == IPC_PRIVATE {
        // IPC get private
        if SHARED_MEMORY_MANAGER.mem_map.lock().len() > 4096 {
            return Err(LinuxError::ENOSPC);
        }
        if size == 0 {
            return Err(LinuxError::EINVAL);
        }
    }
}

pub fn sys_dup3(old_fd: c_int, new_fd: c_int, flags: c_int) -> LinuxResult<isize> {
    let old_file_like = fd_lookup(old_fd as _);
    if old_fd == new_fd {
        // If old_fd equals new_fd, then dup3() fails with the error EINVAL
        return Err(LinuxError::EINVAL);
    }
}
```

ERRORS top

EBADF *olddfd* isn't an open file descriptor.

EBADF *newfd* is out of the allowed range for file descriptors (see the discussion of `RLIMIT_NOFILE` in `getrlimit(2)`).

EBUSY (Linux only) This may be returned by `dup2()` or `dup3()` during a race condition with `open(2)` and `dup()`.

EINTR The `dup2()` or `dup3()` call was interrupted by a signal; see `signal(7)`.

EINVAL (`dup3()`) *flags* contain an invalid value.

EINVAL (`dup3()`) *olddfd* was equal to *newfd*.

EMFILE The per-process limit on the number of open file descriptors has been reached (see the discussion of `RLIMIT_NOFILE` in `getrlimit(2)`).

ENOMEM Insufficient kernel memory was available.

ERRORS top

EACCES An attempt was made to change the process group ID of one of the children of the calling process and the child had already performed an `execve(2)` (`setpgid()`, `setpgrp()`).

EINVAL *pgid* is less than 0 (`setpgid()`, `setpgrp()`).

EPERM An attempt was made to move a process into a process group in a different session, or to change the process group ID of one of the children of the calling process and the child was in a different session, or to change the process group ID of a session leader (`setpgid()`, `setpgrp()`).

EPERM The target process group does not exist. (`setpgid()`, `setpgrp()`).

ESRCH For `getpgid()`: *pid* does not match any process. For `setpgid()`: *pid* is not the calling process and not a child of the calling process.

ERRORS top

EACCES The user does not have permission to access the shared memory segment, and does not have the `CAP_IPC_OWNER` capability in the user namespace that governs its IPC namespace.

EEXIST `IPC_CREAT` and `IPC_EXCL` were specified in *shmflg*, but a shared memory segment already exists for *key*.

EINVAL A new segment was to be created and *size* is less than `SHMMIN` or greater than `SHMMAX`.

EINVAL A segment for the given *key* exists, but *size* is greater than the size of that segment.

ENFILE The system-wide limit on the total number of open files has been reached.

ENOENT No segment exists for the given *key*, and `IPC_CREAT` was not specified.

ENOMEM No memory could be allocated for segment overhead.

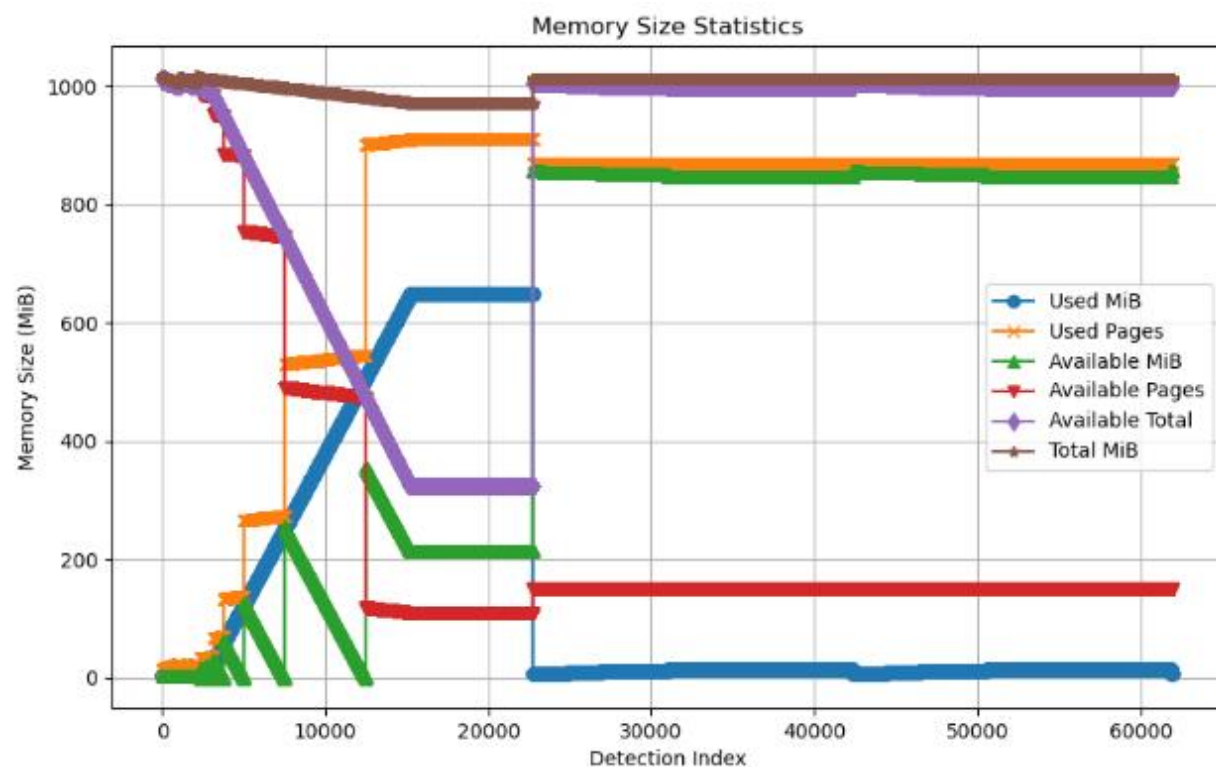
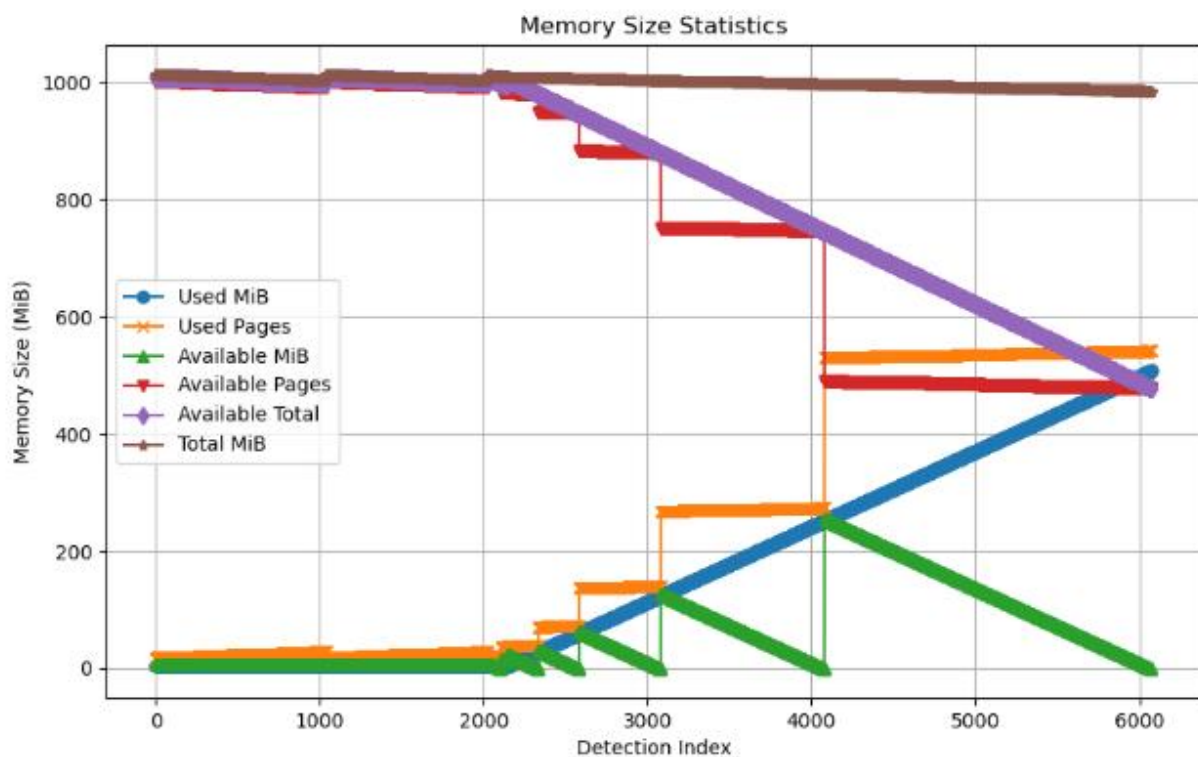
ENOSPC All possible shared memory IDs have been taken (`SHMNI`), or allocating a segment of the requested *size* would cause the system to exceed the system-wide limit on shared memory (`SHMALL`).

我参与解决的问题：libctest测例（内存分配策略）

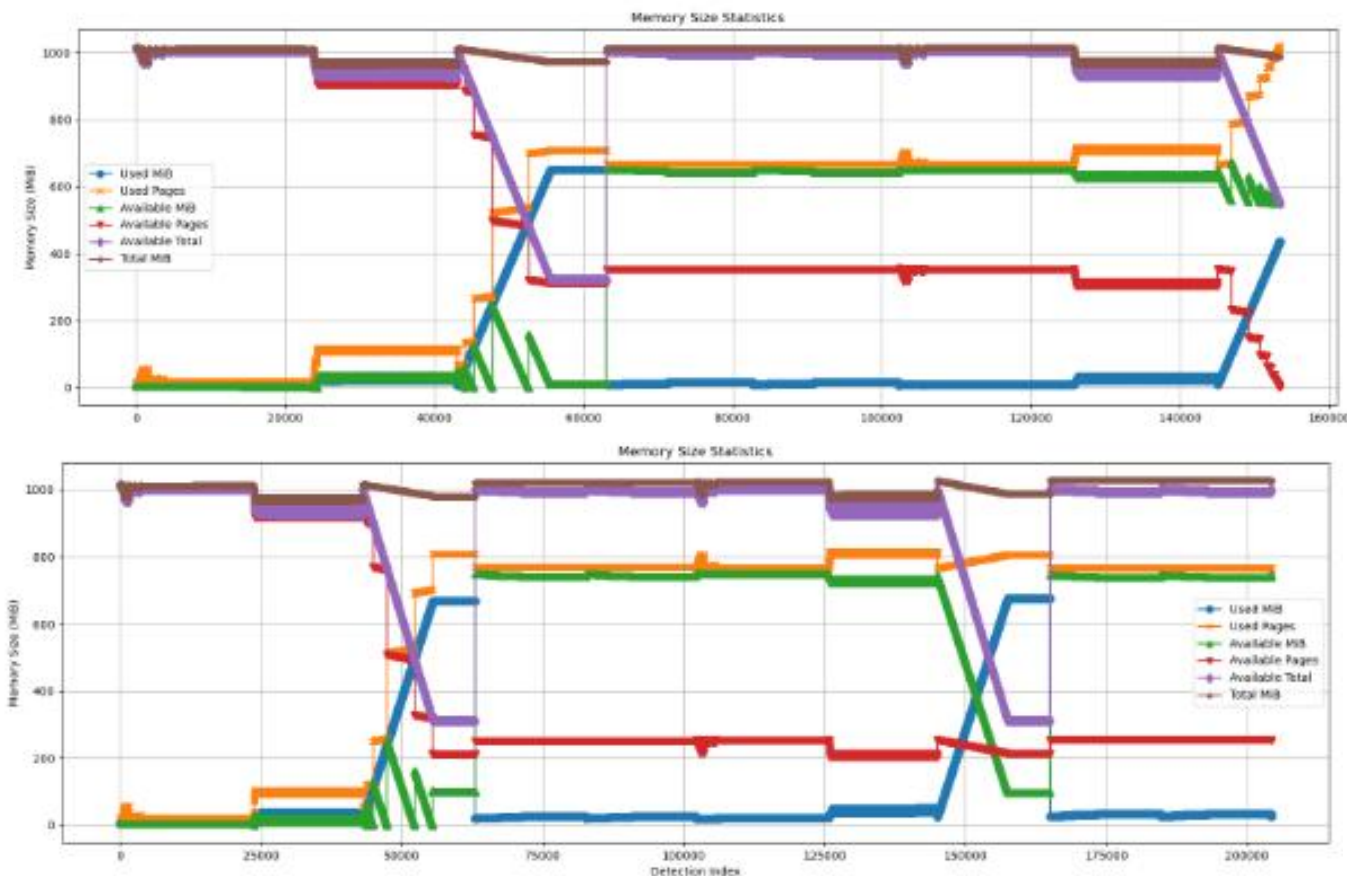
- 在libctest测例的调试过程中，前期总是出现内存分配不足的情况，最开始以为是OS内核存在内存泄露的情况，但测例中内存的分配次数太多，难以进行详细的检查。
- 我编写了可视化的插件，可以对于两级内存分配器(页分配器和字节分配器)的分配情况进行可视化展示，帮助更直观地定位问题进行调试。
- 最终Undefined-OS成功通过了libctest测例。

修改分配策略

- 线上测评平台的运行命令限制了我们的OS可以使用的最大内存空间。
- 基于可视化的结果，2倍增长策略在最后一次试图分配的大小过大，导致了内存不足，如左图。
- 所以调整了内存分配策略，在页不足的情况下不再固定申请分配上次两倍的大小，而是在内存剩余大小少于一定值的情况下申请剩余大小的一定比例，成功解决了问题，如右图。



修改分配策略



- 后续又发现如果在跑过一遍musl的libctest之后再进行一次glibc的libctest，即连续进行两次libctest，还是会出现内存不足的问题，如上图。
- 可以发现当前内存系统在当前字节分配器有足够available MiB的情况下依然试图申请新的page，导致了内存不足。我们对内存分配策略进行了更换，把tlf改成slab解决了问题，如下图。
- 原因最开始推测可能是tlf策略更容易产生外部碎片，总内存足够，但被分割成多个小块，没有足够大的连续空闲块，无法满足当前申请的大小，只能继续申请新的页。
- 但按理来说tlf策略在可变对象的情境下避免碎片的能力强于slab，后续发现了是Starry的原始仓库的tlf实现有问题，只改变了注册信息没有真正释放内存。
- 实际上我们开发过程中常会遇到Starry的遗留bug，这是不可避免的，往往这种错误还更难以排查。

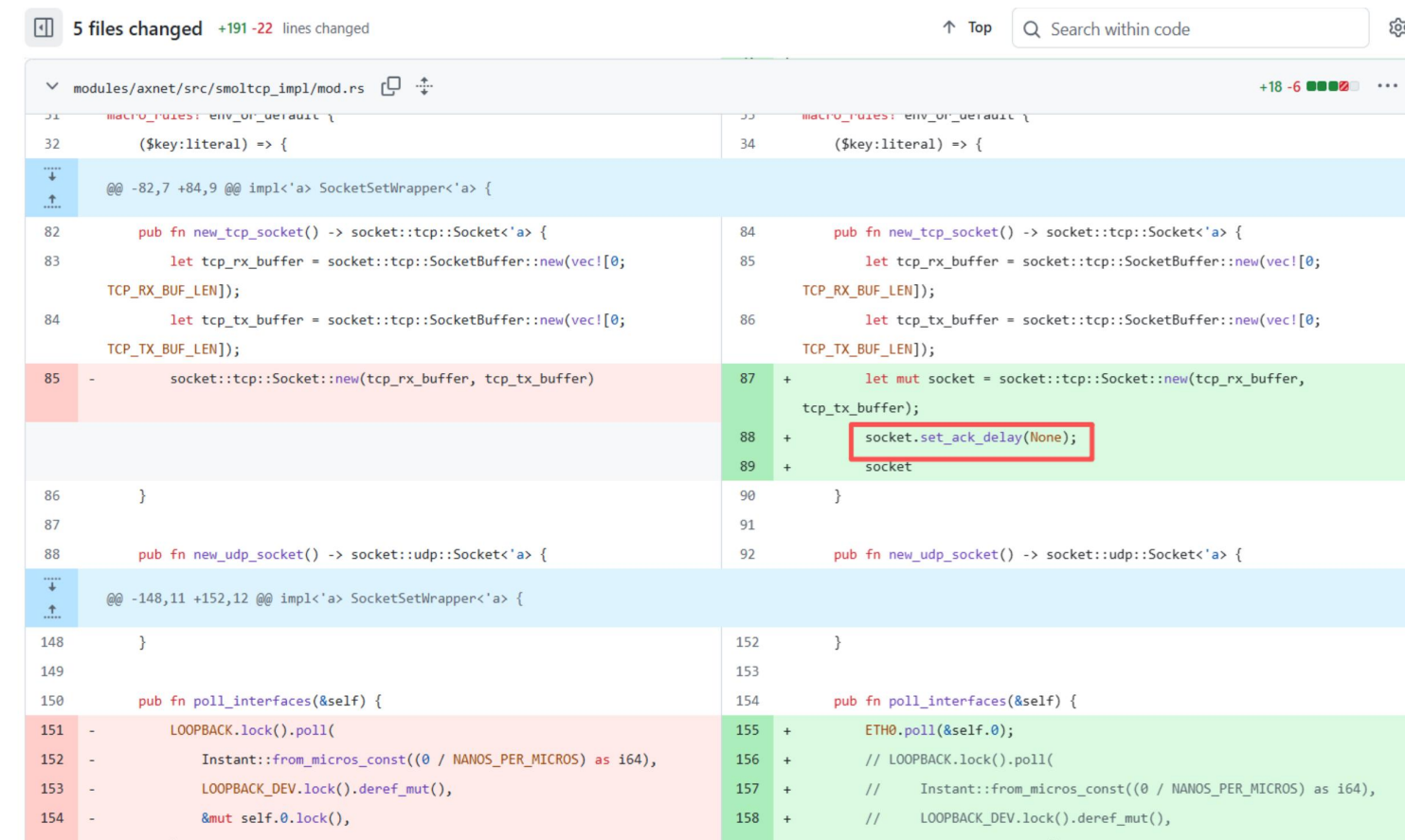
我参与实现的模块：Undefined-OS网络模块

- Undefined OS 的网络模块是基于 smoltcp 库（一个嵌入式TCP/IP 协议栈）实现的，设计目标是提供高效、灵活且易于扩展的网络通信能力。
- Undefined OS 的网络模块提供了对 IPv4 和 IPv6 的支持，实现了 POSIX 风格的TCP/UDP socket API，并且支持 DNS 查询。支持 socket 的基础功能，包括套接字创建、绑定、监听、连接、发送和接收数据等，支持本地(local/unix)套接字和网络套接字。
- 网络模块的功能主要通过套接字提供，根据 Unix 的一切都是文件的设计理念，网络模块将套接字抽象为文件描述符，提供统一的文件操作接口。内部实现对smoltcp 库的封装，提供了对网络设备的抽象和管理。网络模块支持多种地址族，包括 AF_INET、AF_INET6 等，能够处理 IPv4 和 IPv6 双协议栈，实现了TCP 和 UDP传输协议。

我参与解决的问题：现场赛git网络测例（克隆gitee/github仓库）

- 我们通过WireShark抓包发现在tcp三次握手之后，传输数据过程中我们的OS没有及时发回数据包的ACK包，导致远程server(gitee.com)不确定我们的OS是否收到数据包，一直触发重传，最终导致仓库克隆失败，后续我们把实现改为即时发回ACK包，解决问题。
- 原本仅仅支持本地回环，后续添加使用以太网卡进行发包，真实地向远程服务器发送数据包。

```
let iface = if remote_endpoint.addr.as_bytes()[0] == 127 {  
    super::LOOPBACK.try_get().unwrap()  
} else {  
    info!("Use eth net");  
    &super::ETH0.iface  
};
```



我参与解决的问题：stat系列syscall的多架构支持

```
struct stat {  
    dev_t st_dev;  
    ino_t st_ino;  
    mode_t st_mode;  
    nlink_t st_nlink;  
    uid_t st_uid;  
    gid_t st_gid;  
    dev_t st_rdev;  
    unsigned long __pad;  
    off_t st_size;  
    blksize_t st_blksize;  
    int __pad2;  
    blkcnt_t st_blocks;  
    struct timespec st_atim;  
    struct timespec st_mtim;  
    struct timespec st_ctim;  
    unsigned __unused[2];  
};
```

x86_64

```
4 struct stat {  
5     dev_t st_dev;  
6     ino_t st_ino;  
7     nlink_t st_nlink;  
8+  
9+     mode_t st_mode;  
10    uid_t st_uid;  
11    gid_t st_gid;  
12+    unsigned int __pad0;  
13    dev_t st_rdev;  
14    off_t st_size;  
15    blksize_t st_blksize;  
16    blkcnt_t st_blocks;  
17+  
18    struct timespec st_atim;  
19    struct timespec st_mtim;  
20    struct timespec st_ctim;  
21+    long __unused[3];  
22 };
```

riscv64

- 我们发现不同的架构下UserStat结构体在不同架构表现为字段顺序不同，字段大小不同，结构体总长度不同，图中的子类型长度也不一致，例如mode_t的长度也不相同，最初仅仅支持x86_64架构导致了错位，之后我们添加了全架构正确的结构体之后问题解决。
- 这也给我们一些开发多架构系统调用一些经验和启示，不同架构的各项细节设置存在不同，需要为其分别提供支持。

开源社区贡献

- 作品开源：本次参赛作品代码已经开源在比赛官网上，欢迎之后其他同学或小组基于我们的Undefined-OS进行开发，继续参加OS大赛
- Undefined代码仓库链接：<https://github.com/eternalcomet/undefined-os>
- 文档编写：参与编写了OS大赛赛题协作文档和Starry-Tutorial-Book部分内容，同时编写了Undefined-OS自己的项目文档
- Undefined文档链接：<https://undefined-os.github.io/doc/>
- 后续计划更详细地总结一份比赛全过程的经验文档，帮助后续同学参赛

未来方向

- Undefined-OS在决赛现场赛测例排行榜中排名第三，我总结了如下待完善的功能和支持。
- Loongarch上板
- Riscv上板的网络功能（网卡驱动）
- rustc编译器仅能输出-h提示信息，无法正确编译并运行rust代码，这是实现“OS自举”的关键一环



清华大学
Tsinghua University

感谢聆听！

宋建昊 Undefined-OS队

2025年09月23日