GalOS 项目工程二：系统调用完善

> 增强 GalOS 的系统调用支持，提升与 Linux 的兼容性

目录

Part1：项目背景

1.1 项目目标

工程二旨在进一步增强 GalOS 的系统调用支持，使其更接近 Linux 系统调用规范。主要聚焦于：

1. **完善系统调用接口**：提升与 Linux 的兼容性
2. **增强内存管理功能**：提升系统性能和安全性
3. **确保与 ArceOS 的无缝集成**
4. **保持代码的可维护性和可扩展性**

1.2 技术挑战

挑战领域	具体问题	解决方案
内存管理复杂性	Linux 内存管理语义复杂，涉及多种页面状态	设计清晰的 Backend 抽象层
性能要求	系统调用频繁，需要高效实现	使用 Rust 零成本抽象
兼容性	需要严格遵循 Linux 系统调用规范	参考 Linux 手册页和 glibc 实现
架构适配	ArceOS 的内存模型与 Linux 不完全一致	在 AddrSpace 层进行适配

1.3 项目规模

- **新增系统调用**：6 个（madvise, msync, mlock, mlock2, setfsuid, setfsgid）
- **修改模块**：AddrSpace、Backend trait 及其 4 个实现
- **代码文件**：涉及 `api/src/syscall/mm/mmap.rs`、`api/src/syscall/sys.rs`、`arceos/modules/axmm` 等
- **实现行数**：约 800+ 行新增代码

Part2：新增系统调用

2.1 内存建议系统调用

2.1.1 `sys_madvise`

功能：向内核提供关于内存区域访问模式的建议，以优化内存管理。

函数签名：

```
pub fn sys_madvise(addr: usize, length: usize, advice: u32) -> AxResult<isize>
```

参数说明：

参数	类型	说明
`addr`	`usize`	内存区域起始地址（必须 4K 对齐）
`length`	`usize`	内存区域长度
`advice`	`u32`	访问模式建议

支持的 `advice` 参数：

常量	值	说明
`MADV_NORMAL`	0	重置之前的建议，使用默认行为
`MADV_RANDOM`	1	预期随机访问，释放超过阈值的未使用页面
`MADV_SEQUENTIAL`	2	预期顺序访问，执行预取优化
`MADV_WILLNEED`	3	预加载内存页到物理内存
`MADV_DONTNEED`	4	标记内存页可回收
`MADV_REMOVE`	9	移除内存页

实现示例：

```
// api/src/syscall/mm/mmap.rs
pub fn sys_madvise(addr: usize, length: usize, advice: u32) -> AxResult<isize> {
    debug!("sys_madvise <= addr: {addr:#x}, length: {length:x}, advice: {advice:#x}");

    // 验证地址对齐和长度
    if addr % PageSize::Size4K as usize != 0 || length == 0 {
        return Err(AxError::InvalidInput);
    }

    let curr = current();
    let mut aspace = curr.as_thread().proc_data.aspace.lock();
    let start_addr = VirtAddr::from(addr);
    let length = align_up_4k(length);

    // 检查内存区域是否存在
    if !aspace.contains_range(start_addr, length) {
        return Err(AxError::InvalidInput);
    }

    // 根据advice参数实现不同的策略
    match advice {
        MADV_NORMAL => {
            // 默认行为，重置之前的建议
            debug!("MADV_NORMAL: Reset memory access hints");
        },
```

```
        MADV_RANDOM => {
            debug!("MADV_RANDOM: Expecting random memory access");

            // 仅释放超过一定阈值的未使用页面，保留部分工作集
            const KEEP_THRESHOLD: usize = 5 * PageSize::Size4K as usize; // 保留最近使用的5页
            if length > KEEP_THRESHOLD {
                // 释放除了最近使用页面之外的其他页面
                // 这里需要实现页面使用时间的跟踪
                aspace.clear_area(start_addr + KEEP_THRESHOLD, length - KEEP_THRESHOLD)?;
            }
        },
        MADV_SEQUENTIAL => {
            // 顺序访问模式，优化预取
            debug!("MADV_SEQUENTIAL: Expecting sequential memory access");

            // 基本预取：加载当前区域
            aspace.populate_area(start_addr, length, MappingFlags::READ)?;

            // 高级预取：尝试加载后续区域（可选）
            // 这里可以根据需要调整预取的额外长度
            const PREFETCH_EXTENSION: usize = 10 * PageSize::Size4K as usize; // 预取额外的10个页面

            let current_end = start_addr + length;
            let _prefetch_end = current_end + PREFETCH_EXTENSION;

            // 检查预取区域是否在地址空间范围内
            if aspace.contains_range(current_end, PREFETCH_EXTENSION) {
                // 尝试预取后续区域，但忽略错误（如果内存不足等情况）
                let _ = aspace.populate_area(current_end, PREFETCH_EXTENSION, MappingFlags::READ);
            }
        },
        MADV_WILLNEED => {
            // 预加载内存页
            debug!("MADV_WILLNEED: Preloading memory pages");
            aspace.populate_area(start_addr, length, MappingFlags::READ)?;
        },
        MADV_DONTNEED => {
            // 释放内存页但保留地址空间
            debug!("MADV_DONTNEED: Releasing memory pages");
            // 实现释放页面的逻辑
            aspace.clear_area(start_addr, length)?;
        },
        MADV_REMOVE => {
                    // 从映射中删除页面
                    debug!("MADV_REMOVE: Removing pages from mapping");
                    // 目前只支持匿名映射的页面删除
                    // 对于文件映射，需要更复杂的处理
                    if let Some(area) = aspace.find_area(start_addr) {
                        // 检查是否为匿名映射（简化实现）
                        match area.backend().page_size() {
                            PageSize::Size4K => {
                                // 对于匿名映射，我们可以直接解除映射
                                aspace.unmap(start_addr, length)?;
                            },
                            _ => {
                                // 对于其他类型的映射，返回错误
                                warn!("MADV_REMOVE only supported for anonymous mappings");
                                return Err(AxError::OperationNotSupported);
                            }
                        }
                    }
                },
        MADV_DONTFORK => {
            // 子进程不继承此内存区域
            debug!("MADV_DONTFORK: Child processes won't inherit this memory");
            aspace.set_dontfork(start_addr, length)?;
        },
        MADV_DOFORK => {
            // 重置MADV_DONTFORK标志
            debug!("MADV_DOFORK: Child processes will inherit this memory");
            aspace.set_dofork(start_addr, length)?;
        },
        // 其他建议类型的实现...
        _ => {
```

```
            warn!("Unknown madvise advice: {advice}");
            return Err(AxError::InvalidInput);
        },
    }

    Ok(0)
}
```

实现位置：`api/src/syscall/mm/mmap.rs`

2.1.2 `sys_msync`

功能：将内存映射中的修改同步回后备存储（如文件）。

函数签名：

```
pub fn sys_msync(addr: usize, length: usize, flags: u32) -> AxResult<isize>
```

参数说明：

参数	类型	说明
`addr`	`usize`	内存区域起始地址（必须 4K 对齐）
`length`	`usize`	内存区域长度
`flags`	`u32`	同步标志

支持的 `flags` 参数：

常量	值	说明
`MS_SYNC`	1	同步等待，直到所有修改写入磁盘
`MS_ASYNC`	2	异步同步，不等待完成
`MS_INVALIDATE`	4	使其他进程的映射失效（当前未实现）

实现示例：

```
// api/src/syscall/mm/mmap.rs
const MS_SYNC: u32 = 1;
const MS_ASYNC: u32 = 2;
const MS_INVALIDATE: u32 = 4;

pub fn sys_msync(addr: usize, length: usize, flags: u32) -> AxResult<isize> {
    debug!("sys_msync <= addr: {addr:#x}, length: {length:x}, flags: {flags:#x}");

    // 检查addr是否对齐
    if !is_aligned_4k(addr) {
        return Err(AxError::InvalidInput);
    }

    // 检查flags是否有效
    if flags & !(MS_SYNC | MS_ASYNC | MS_INVALIDATE) != 0 {
        return Err(AxError::InvalidInput);
    }
```

```
    // 检查MS_SYNC和MS_ASYNC是否同时设置
    if (flags & MS_SYNC) != 0 && (flags & MS_ASYNC) != 0 {
        return Err(AxError::InvalidInput);
    }

    // 获取当前任务的地址空间
    let current = current();
    let mut aspace = current.as_thread().proc_data.aspace.lock();

    // 同步内存区域
    let start = VirtAddr::from(addr);
    aspace.sync_area(start, length)?;

    // 处理MS_INVALIDATE标志（可选，当前实现中暂不处理）
    if flags & MS_INVALIDATE != 0 {
        // TODO: 实现缓存失效功能
    }

    Ok(0)
}
```

实现位置：`api/src/syscall/mm/mmap.rs`

2.1.3 `sys_mlock` / `sys_mlock2`

功能：将内存区域锁定在物理内存中，防止被交换出去。

函数签名：

```
pub fn sys_mlock(addr: usize, length: usize) -> AxResult<isize>
pub fn sys_mlock2(addr: usize, length: usize, flags: u32) -> AxResult<isize>
```

参数说明：

参数	类型	说明
`addr`	`usize`	内存区域起始地址
`length`	`usize`	内存区域长度
`flags`	`u32`	锁定标志（当前仅支持 0）

实现示例：

```
// api/src/syscall/mm/mmap.rs
/// 锁定内存区域，防止被交换出去
pub fn sys_mlock(addr: usize, length: usize) -> AxResult<isize> {
    sys_mlock2(addr, length, 0)
}

/// 带有额外标志的内存锁定函数
pub fn sys_mlock2(addr: usize, length: usize, flags: u32) -> AxResult<isize> {
    debug!("sys_mlock2 <= addr: {addr:#x}, length: {length:x}, flags: {flags:#x}");

    // 检查flags是否合法
    if flags != 0 {
        // 当前只支持flags=0
        return Err(AxError::InvalidInput);
    }

    // 确保长度不为0
    if length == 0 {
```

```
            return Err(AxError::InvalidInput);
        }

        // 获取当前任务
        let curr = current();
        let mut aspace = curr.as_thread().proc_data.aspace.lock();

        // 对齐地址和长度到4K页面
        let start = addr.align_down_4k();
        let end = (addr + length).align_up_4k();
        let aligned_length = end - start;

        // 验证内存区域是否在地址空间范围内
        let start_addr = VirtAddr::from(start);
        if !aspace.contains_range(start_addr, aligned_length) {
            return Err(AxError::InvalidInput);
        }

        // 验证内存区域是否可访问
        if !aspace.can_access_range(start_addr, aligned_length, MappingFlags::READ | MappingFlags::WRITE) {
            return Err(AxError::InvalidInput);
        }

        // 使用populate_area功能将内存锁定在物理内存中
        // 这会将所有页面映射到物理内存，并防止它们被交换出去
        aspace.populate_area(start_addr, aligned_length, MappingFlags::READ | MappingFlags::WRITE)?;

        Ok(0)
    }
```

实现位置：`api/src/syscall/mm/mmap.rs`

Part3：内存管理增强

3.1 AddrSpace 新增函数

在 `AddrSpace` 结构中新增了 4 个关键函数，用于支持新的系统调用：

3.1.1 `clear_area`

功能：清除指定内存区域的映射，但保留地址空间。

函数签名：

```
pub fn clear_area(&mut self, start: usize, size: usize) -> AxResult<()>
```

参数说明：

参数	类型	说明
`start`	`usize`	内存区域起始地址
`size`	`usize`	内存区域大小

实现逻辑：

```
// arceos/modules/axmm/src/aspace.rs
    pub fn clear_area(&mut self,mut start: VirtAddr, size: usize) -> AxResult {
```

```
        self.validate_region(start, size)?;
        let end = start + size;

        let mut modify = self.pt.modify();
        while let Some(area) = self.areas.find(start) {
            let range = VirtAddrRange::new(start, area.end().min(end));
            area.backend()
                .clear(range, area.flags(), &mut modify)?;
            start = area.end();
            assert!(start.is_aligned_4k());
            if start >= end {
                break;
            }
        }

        Ok(())
    }
```

实现位置：`arceos/modules/axmm/src/aspace.rs`

3.1.2 `set_dontfork`

功能：标记指定内存区域在 `fork` 时不被复制。

函数签名：

```
pub fn set_dontfork(&mut self, start: usize, size: usize) -> AxResult<()>
```

实现逻辑：

```
// arceos/modules/axmm/src/aspace.rs
    pub fn set_dontfork(&mut self, start: VirtAddr, size: usize) -> AxResult {
        self.validate_region(start, size)?;
        let end = start + size;
        self.dontfork_areas.insert(start, end);
        Ok(())
    }
```

实现位置：`arceos/modules/axmm/src/aspace.rs`

3.1.3 `set_dofork`

功能：取消标记指定内存区域在 `fork` 时不被复制的设置。

函数签名：

```
pub fn set_dofork(&mut self, start: usize, size: usize) -> AxResult<()>
```

实现逻辑：

```
// arceos/modules/axmm/src/aspace.rs
    pub fn set_dofork(&mut self, start: VirtAddr, size: usize) -> AxResult {
        self.validate_region(start, size)?;
        let end = start + size;
```

```
        let mut to_remove = Vec::new();
        let mut to_insert = Vec::new();

        for (range_start, range_end) in &self.dontfork_areas {
            let range = VirtAddrRange::new(*range_start, *range_end);
            let target = VirtAddrRange::new(start, end);

            if range.overlaps(target){
                continue;
            }

            to_remove.push(*range_start);

            // 处理重叠区域的前后部分
            if *range_start < start {
                to_insert.push((*range_start, start));
            }
            if *range_end > end {
                to_insert.push((end, *range_end));
            }
        }

        // 移除旧的区域
        for start_addr in to_remove {
            self.dontfork_areas.remove(&start_addr);
        }

        // 插入新的区域
        for (new_start, new_end) in to_insert {
            self.dontfork_areas.insert(new_start, new_end);
        }

        Ok(())
    }
```

实现位置：`arceos/modules/axmm/src/aspace.rs`

3.1.4 `sync_area`

功能：将指定内存区域的修改同步回后备存储。

函数签名：

```
pub fn sync_area(&mut self, start: usize, size: usize) -> AxResult<()>
```

实现逻辑：

```
// arceos/modules/axmm/src/aspace.rs
    pub fn sync_area(&mut self, mut start: VirtAddr, size: usize) -> AxResult {
        self.validate_region(start, size)?;
        let end = start + size;

        let mut modify = self.pt.modify();
        while let Some(area) = self.areas.find(start) {
            let range = VirtAddrRange::new(start, area.end().min(end));
            area.backend()
                .sync(range, area.flags(), &mut modify)?;
            start = area.end();
            assert!(start.is_aligned_4k());
            if start >= end {
                break;
            }
        }
```

```
        Ok(())
    }
```

实现位置：`arceos/modules/axmm/src/aspace.rs`

3.2 Backend Trait 扩展

在 `Backend` trait 中新增了两个方法，所有后端都需要实现：

3.2.1 `clear` 方法

功能：清除内存区域的映射。

方法签名：

```
fn clear(
    &self,
    range: VirtAddrRange,
    flags: MappingFlags,
    pt: &mut impl PagingIf,
) -> AxResult<()>
```

参数说明：

参数	类型	说明
`range`	`VirtAddrRange`	虚拟地址范围
`flags`	`MappingFlags`	映射标志
`pt`	`&mut impl PagingIf`	页表修改器

默认实现：

```
fn clear(
    &self,
    range: VirtAddrRange,
    flags: MappingFlags,
    pt: &mut impl PagingIf,
) -> AxResult<()> {
    // 默认实现：仅解除映射
    pt.unmap(range)?;
    Ok(())
}
```

3.2.2 `sync` 方法

功能：将内存区域的修改同步回后备存储。

方法签名：

```
fn sync(
    &self,
    range: VirtAddrRange,
```

```
    flags: MappingFlags,
    pt: &mut impl PagingIf,
) -> AxResult<()>
```

默认实现：

```
fn sync(
    &self,
    range: VirtAddrRange,
    flags: MappingFlags,
    pt: &mut impl PagingIf,
) -> AxResult<()> {
    // 默认实现：无操作
    Ok(())
}
```

Part4：后端实现

4.1 CowBackend (Copy-on-Write)

用途：支持写时复制的内存区域，主要用于 `fork` 后的父子进程共享。

`clear` 实现

```
impl Backend for CowBackend {
    fn clear(
        &self,
        range: VirtAddrRange,
        flags: MappingFlags,
        pt: &mut impl PagingIf,
    ) -> AxResult<()> {
        // 解除页面映射
        pt.unmap(range)?;

        // 减少物理帧引用计数
        for page in range.pages() {
            if let Some(paddr) = pt.query(page.start_address()) {
                let frame = PhysFrame::from_paddr(paddr);
                frame.dec_ref();

                // 当引用计数为 0 时释放帧
                if frame.ref_count() == 0 {
                    deallocate_frame(frame);
                }
            }
        }

        Ok(())
    }

    fn sync(&self, ...) -> AxResult<()> {
        // CowBackend 无需同步（使用默认实现）
        Ok(())
    }
}
```

特点：

- 解除映射时需要管理引用计数
- 无需同步操作（纯内存后端）

4.2 FileBackend

用途：支持文件映射（`mmap` 文件），需要将脏页写回文件。

`clear` 实现

```
impl Backend for FileBackend {
    fn clear(
        &self,
        range: VirtAddrRange,
        flags: MappingFlags,
        pt: &mut impl PagingIf,
    ) -> AxResult<()> {
        // 解除页面映射
        pt.unmap(range)?;
        Ok(())
    }

    fn sync(
        &self,
        range: VirtAddrRange,
        flags: MappingFlags,
        pt: &mut impl PagingIf,
    ) -> AxResult<()> {
        // 调用 CachedFile 的 sync 方法将缓存脏页写回文件
        self.file.sync(range)?;
        Ok(())
    }
}
```

特点：

- `clear` 仅解除映射
- `sync` 会将修改写回文件

4.3 LinearBackend

用途：线性映射，通常用于内核地址空间。

实现

```
impl Backend for LinearBackend {
    fn clear(
        &self,
        range: VirtAddrRange,
        flags: MappingFlags,
        pt: &mut impl PagingIf,
    ) -> AxResult<()> {
        // 解除页面映射
        pt.unmap(range)?;
        Ok(())
    }

    fn sync(&self, ...) -> AxResult<()> {
        // LinearBackend 无需同步（使用默认实现）
        Ok(())
    }
}
```

特点：

- 简单的解除映射
- 无需同步操作

4.4 SharedBackend

用途：共享内存区域，多个进程可以访问。

实现

```
impl Backend for SharedBackend {
    fn clear(
        &self,
        range: VirtAddrRange,
        flags: MappingFlags,
        pt: &mut impl PagingIf,
    ) -> AxResult<()> {
        // 解除页面映射
        pt.unmap(range)?;
        Ok(())
    }

    fn sync(&self, ...) -> AxResult<()> {
        // SharedBackend 无需同步（使用默认实现）
        Ok(())
    }
}
```

特点：

- 解除映射但不释放物理页
- 无需同步操作（共享内存不持久化）

Part5：代码组织

5.1 系统调用层

```
api/src/syscall/
├── mm/
│   └── mmap.rs              # sys_madvise, sys_msync, sys_mlock, sys_mlock2
├── sys.rs                   # sys_setfsuid, sys_setfsgid
├── fs/                      # 文件系统相关系统调用
│   ├── io.rs
│   ├── fd_ops.rs
│   ├── stat.rs
│   └── ...
├── task/                    # 进程/线程相关系统调用
│   ├── clone.rs
│   ├── execve.rs
│   └── ...
├── net/                     # 网络相关系统调用
├── sync/                    # 同步原语系统调用
└── mod.rs                   # 系统调用注册和分发
```

5.2 内存管理层

```
arceos/modules/axmm/
├── src/
│   ├── aspace.rs            # AddrSpace 实现（新增 4 个方法）
│   ├── backend/
│   │   ├── mod.rs           # Backend trait 定义（新增 2 个方法）
│   │   ├── cow.rs           # CowBackend 实现
│   │   ├── file.rs          # FileBackend 实现
│   │   ├── linear.rs        # LinearBackend 实现
│   │   └── shared.rs        # SharedBackend 实现
│   └── ...
```

5.3 系统调用注册

在 `api/src/syscall/mod.rs` 中注册新系统调用：

```rust
pub fn handle_syscall(uctx: &mut UserContext) {
    let sysno = Sysno::new(uctx.sysno());

    let result = match sysno {
        // ... 其他系统调用

        // 内存管理系统调用
        Sysno::madvise => sys_madvise(uctx.arg0(), uctx.arg1() as _, uctx.arg2() as _),
        Sysno::msync => sys_msync(uctx.arg0(), uctx.arg1(), uctx.arg2() as _),
        Sysno::mlock => sys_mlock(uctx.arg0(), uctx.arg1()),
        Sysno::mlock2 => sys_mlock2(uctx.arg0(), uctx.arg1(), uctx.arg2() as _),

        // 文件系统 ID 管理
        Sysno::setfsuid => sys_setfsuid(uctx.arg0() as _),
        Sysno::setfsgid => sys_setfsgid(uctx.arg0() as _),

        // ... 更多系统调用
    };

    uctx.set_retval(result);
}
```

Part6：TODO

6.1 功能扩展

🟡 高优先级

- [ ] **MADV_INVALIDATE 支持**：实现 `msync` 的 MS_INVALIDATE 标志
- [ ] **mlock2 高级特性**：支持 MLOCK_ONFAULT 标志
- [ ] **内存使用统计**：记录每个进程的内存锁定量，实施 RLIMIT_MEMLOCK 限制

🟢 中优先级

- [ ] **更多 madvise 选项**：支持 MADV_MERGEABLE、MADV_HUGEPAGE 等
- [ ] **munlock 实现**：支持解除内存锁定
- [ ] **mmap2 系统调用**：支持大于 2GB 的文件偏移

🔵 低优先级

- [ ] **NUMA 支持**：支持 MADV_PREFERRED_MBIND 等 NUMA 相关建议
- [ ] **透明大页**：优化大内存分配性能
- [ ] **内存压缩**：支持 MADV_WILLNEED 的智能预测

6.2 性能优化

- [ ] **批量解除映射**：优化 `clear_area` 的页表操作
- [ ] **异步同步**：实现真正的 MS_ASYNC 异步写回
- [ ] **缓存预取**：优化 MADV_SEQUENTIAL 的预取策略

6.3 兼容性

- [ ] **更多架构支持**：验证在 LoongArch64 上的功能
- [ ] **musl libc 兼容**：确保与 musl libc 的系统调用包装器兼容
- [ ] **strace 支持**：添加系统调用跟踪支持

附录A：系统调用统计

A.1 原StarryOS 系统调用总览

类别	已实现数量	说明
文件系统	35+	open, read, write, close, stat, mount 等
进程管理	20+	fork, execve, wait, exit, clone 等
内存管理	15+	mmap, munmap, brk, madvise, msync, mlock 等
网络	25+	socket, bind, connect, send, recv 等
同步	10+	futex, membarrier 等
信号	15+	sigaction, sigprocmask, kill 等
I/O 多路复用	6+	select, poll, epoll 系列
IPC	5+	shm 系列
其他	10+	uname, setfsuid, setfsgid 等
总计	**140+**	持续增加中

A.2 工程二新增系统调用

系统调用	编号	实现文件	行数
`sys_madvise`	28	`api/src/syscall/mm/mmap.rs`	~80
`sys_msync`	26	`api/src/syscall/mm/mmap.rs`	~50
`sys_mlock`	149	`api/src/syscall/mm/mmap.rs`	~10
`sys_mlock2`	325	`api/src/syscall/mm/mmap.rs`	~40

附录B：参考资料

B.1 Linux 手册页

- `madvise(2)` - https://man7.org/linux/man-pages/man2/madvise.2.html
- `msync(2)` - https://man7.org/linux/man-pages/man2/msync.2.html
- `mlock(2)` - https://man7.org/linux/man-pages/man2/mlock.2.html
- `setfsuid(2)` - https://man7.org/linux/man-pages/man2/setfsuid.2.html

B.2 相关代码

- **Linux 内核源码**：mm/madvise.c, mm/msync.c, mm/mlock.c
- **glibc 实现**：sysdeps/unix/sysv/linux/
- **ArceOS 内存管理**：https://github.com/arceos-org/arceos/tree/main/modules/axmm

---

最后更新：2026-01-03