期中进展汇报

黄雨婕 2022010887

部分 syscall 实现

poll

- 监视一组文件描述符，阻塞等待直到以下情况之一发生：
  - 至少一个文件描述符就绪（可读/可写/异常）。
  - 超时。
  - 被信号中断（如 `SIGINT`）。

```
///
/// # 参数
/// - `fds`: 需要监控的文件描述符
/// - `timeout`: 超时时间（以毫秒为单位），负值表示无限等待，0 表示立即返回
///
/// # 返回值
/// - 返回准备就绪的文件描述符数量（≥0）
/// - `Err(LinuxError)`: 如果发生错误，返回 Linux 错误码

pub fn sys_poll_impl(fds: &mut [PollFd], timeout: i32) -> LinuxResult<i32> {
    // 初始化所有文件描述符的返回事件为 0
    for fd in fds.iter_mut() {
        fd.revents = 0;
    }
    // 获取当前时间（以ns为单位）
    let now = axhal::time::monotonic_time_nanos();
    loop {
        let mut updated = false;
        for fd in fds.iter_mut() {
            if fd.fd < 0 {
                // 忽略无效的文件描述符
                continue;
            }
            let f = get_file_like(fd.fd);
            if let Err(_) = f {
                // 如果文件描述符无效，设置返回事件为 POLLNVAL
                fd.revents = POLLNVAL;
                continue;
            }
            if let Some(pipe) = f.clone()?.into_any().downcast_ref::<Pipe>() {
                if pipe.write_end_close() {
                    // 如果管道的写端已关闭，设置返回事件为 POLLHUP
```

```
                    fd.revents |= POLLHUP;
                    updated = true;
                }
            }
            match f?.poll() {
                Ok(state) => {
                    // 如果文件描述符可读且事件包含 POLLIN，设置返回事件为 POLLIN
                    if state.readable && fd.events & POLLIN != 0 {
                        fd.revents |= POLLIN;
                        updated = true;
                    }
                    // 如果文件描述符可写且事件包含 POLLOUT，设置返回事件为 POLLOUT
                    if state.writable && fd.events & POLLOUT != 0 {
                        fd.revents |= POLLOUT;
                        updated = true;
                    }
                }
                Err(_) => {
                    // 如果发生错误（例如管道关闭），设置返回事件为 POLLERR
                    fd.revents = POLLERR;
                    updated = true;
                    continue;
                }
            }
        }
        if updated || timeout == 0 {
            // 如果有更新或超时时间为 0，则退出循环
            break;
        }
        if timeout > 0 {
            let elapsed = axhal::time::monotonic_time_nanos() - now;
            if elapsed >= timeout as u64 * NANOS_PER_MICROS {
                // 如果超时时间已到，退出循环
                break;
            }
        }
        // 暂时让出 CPU
        yield_now();
    }
    let mut updated_count = 0;
    for fd in fds.iter() {
        if fd.revents != 0 {
            // 统计有事件更新的文件描述符数量
            updated_count += 1;
        }
    }
    // 返回更新的文件描述符数量
    Ok(updated_count)
}
```

ppoll：在 `poll` 的基础上扩展，主要改进 **信号处理（屏蔽部分信号） 和 **超时精度（精度更高）。****

```rust
pub fn sys_ppoll(
    fds: UserPtr<PollFd>, // 用户指针，指向需要监控的文件描述符数组
    nfds: c_ulong,        // 文件描述符数组的大小
    timeout: UserConstPtr<timespec>, // 用户指针，指向超时时间的 `timespec` 结构体
    sigmask: UserConstPtr<c_void>,   // 用户指针，指向信号掩码
) -> LinuxResult<isize> {
    // 对用户指针 `fds` 进行类型转换并检查其有效性
    let fds = fds.get_as_array(nfds as _)?;
    let fds: &mut [PollFd] = unsafe { core::slice::from_raw_parts_mut(fds, nfds as
_) };

    // 处理超时时间指针，允许为空
    let timeout = timeout
        .nullable(UserConstPtr::get)?
        .unwrap_or(core::ptr::null());

    // 处理信号掩码指针，允许为空
    let sigmask = sigmask
        .nullable(UserConstPtr::get)?
        .unwrap_or(core::ptr::null());

    // 调用 api::sys_ppoll 函数
    Ok(api::sys_ppoll(fds, timeout, sigmask) as _)
}
```

pread64：从文件描述符的指定偏移量读取数据

```rust
/// 从文件描述符的指定偏移量读取数据
///
/// # 参数
/// - fd: 文件描述符
/// - buf: 指向目标缓冲区的指针
/// - count: 要读取的字节数
/// - offset: 偏移量
///
/// # 返回值
/// - 返回读取的字节数，如果发生错误返回错误码
pub fn sys_pread64(
    fd: c_int,
    buf: *mut c_void,
    count: usize,
    offset: ctypes::off_t,
) -> ctypes::ssize_t {
    debug!(
        "sys_pread64 <= {} {:#x} {} {}",
        fd, buf as usize, count, offset
    );
    syscall_body!(sys_pread64, {
        if buf.is_null() {
            return Err(LinuxError::EFAULT); // 缓冲区指针为空，返回无效地址错误
        }
        let dst = unsafe { core::slice::from_raw_parts_mut(buf as *mut u8, count)
```

```
};
        #[cfg(feature = "fd")]
        {
            //// 如果启用了 fd 功能，使用文件操作
            let file = File::from_fd(fd)?; // 从文件描述符获取文件对象
            let file = file.inner();
            let origin_offset = file.lock().seek(SeekFrom::Current(0))?; // 保存当前
偏移量
            file.lock().seek(SeekFrom::Start(offset as _))?; // 设置到指定偏移量
            let result = file.lock().read(dst)?; // 读取数据
            file.lock().seek(SeekFrom::Start(origin_offset))?; // 恢复原始偏移量
            Ok(result as ctypes::ssize_t) // 返回读取的字节数
        }    #[cfg(not(feature = "fd"))]
        {
            // 如果未启用 fd 功能，提供有限的支持
            warn!("[sys_pread64] pread64 is not supported on this platform");
            match fd {
                0 => Ok(super::stdio::stdin().read(dst, offset)? as
ctypes::ssize_t), // 从标准输入读取
                1 | 2 => Err(LinuxError::EPERM), // 标准输出和错误不支持读取
                _ => Err(LinuxError::EBADF), // 无效的文件描述符
            }
        }
    })
}
```

readv：批量读取

```
/// 按批读取
///
/// # 参数
/// - `fd`: 文件描述符
/// - `iov`: 指向 `iovec` 结构体数组的指针
/// - `iocnt`: `iovec` 数组的长度
///
/// # 返回值
/// - 返回读取的字节数，如果发生错误，返回负值表示错误码
pub unsafe fn sys_readv(fd: c_int, iov: *const ctypes::iovec, iocnt: c_int) ->
ctypes::ssize_t {
    debug!("sys_readv <= fd: {}", fd);
    syscall_body!(sys_readv, {
        // 检查 `iocnt` 是否在有效范围内
        if !(0..=1024).contains(&iocnt) {
            return Err(LinuxError::EINVAL); // 返回无效参数错误
        }

        let iovs = unsafe { core::slice::from_raw_parts(iov, iocnt as usize) };
        let mut ret = 0; // 累计读取的字节数

        for iov in iovs.iter() {
            //// 调用 `sys_read` 读取数据
            let result = sys_read(fd, iov.iov_base, iov.iov_len as usize);
            ret += result;
```

```
            //// 如果读取的字节数小于当前 `iovec` 的长度，则停止
            if result < iov.iov_len as isize {
                break;
            }
        }

        // 返回累计读取的字节数
        Ok(ret)
    })
}
```

BUG 修复

关闭文件描述符

在系统调用 sys_exit 和 sys_exit_group 中添加了对文件描述符关闭的处理逻辑。

```
/// 关闭属于当前进程的所有文件描述符
///
/// # 行为
/// - 遍历文件描述符表并移除所有文件描述符。
pub fn close_all_file_like() {
    // 获取文件描述符表的写锁
    let mut fd_table = FD_TABLE.write();

    // 收集所有文件描述符的 ID
    let all_ids: Vec<_> = fd_table.ids().collect();

    //// 遍历所有文件描述符并移除
    for id in all_ids {
        let _ = fd_table.remove(id);
    }
}
```

```
/// 终止调用进程并执行必要的清理操作
///
/// # 参数
/// - `status`: 进程的退出状态码
///
/// # 行为
/// - 唤醒被 `futex` 阻塞并等待 `clear_child_tid` 指针地址的线程（待实现）。
/// - 关闭属于该进程的所有打开的文件描述符。
/// - 使用指定的状态码退出进程。
pub fn sys_exit(status: i32) -> ! {
    // TODO: 唤醒被 `futex` 阻塞并等待 `clear_child_tid` 指针地址的线程。

    //// 关闭属于该进程的所有打开的文件描述符
    close_all_file_like();
```

```
    // 使用指定的状态码退出进程
    axtask::exit(status);
}
```

pipe 读取卡死

原代码实现是直到读到了足够大小的数据才返回（把缓冲区大小当作目标读取大小），现改为读到数据 （read_size > 0）即可返回。

```
//api/arceos_posix_api/src/imp/pipe.rs

dlet mut ring_buffer = self.buffer.lock();
let loop_read = ring_buffer.available_read();
if loop_read == 0 {
--    if self.write_end_close() {
++    if self.write_end_close() || read_size > 0
++    /* || non_block */
++    {
        return Ok(read_size);
    }
    drop(ring_buffer);
```

getdirent64

该函数用于遍历目录，会将结果写到用户提供的结构体，返回值应该是本次写入字节数，调用者会反复调用直到结果大小为 0（表示已经读完目录下所有大小并写完了），但是原来的实现中，每次调用不会记录上次结果写到哪了，因此从来不返回 0，导致死循环。

对该问题进行了修复。

```
current_offset += entry_length;
Ok(current_offset as _)
```