中山大學

SUN YAT-SEN UNIVERSITY

操作系统实验报告

实验五：fork 的实现、并发与锁机制

姓　　名：　刘家祥

学　　号：　23336152

教学班号：　计科二班

专　　业：　计算机科学与技术

院　　系：　计算机学院

2024~2025 学年第二学期

fork 的实现、并发与锁机制

一. fork 的实现

1. 系统调用

首先我们需要为 fork 添加系统调用号，在项目根目录下的 `Cargo.toml` 文件下找到 `syscall_def`：

```
syscall_def = { path = "pkg/syscall", package = "ysos_syscall" }
```

这告诉我们 `ysos_syscall` 的源代码位于 `pkg/syscall` 目录下，并且按照 `Rust` 的约定，一个库包的主要源文件通常是 `src/lib.rs`。在 `pkg/syscall/src/lib.rs` 文件中我们找到了系统调用的定义，里面有一个 `SYscall` 枚举，它定义了所有的系统调用以及对应的编号。在其中插入 `Fork = 58`。

2. 进程管理

接下来完成 fork 的具体实现（补全指导书中的几个 fixme）：

2.1. 首先在 `Stack` 中添加 `fork` 方法，同时将给出的 `clone_range` 函数添加到 `Stack` 的实现中作为复制栈空间的辅助函数。

- 实现思路：
  - 为子进程计算所需的新的栈空间的范围。
  - 为子进程分配并映射新栈。
  - 复制父进程内容到子进程栈。
  - 返回新的 `Stack` 实例。

> **注意：** 关于新栈的分配，文档建议：
> 通过子进程数量、页表引用计数、当前父进程的栈等信息，为子进程分配合适的栈空间。

并给出了一个期望的栈布局。大概说明了子进程的栈通常分配在父进程栈的下方。而 stack_offset_count 参数可能就是用来指示这是第几个子进程，从而计算偏移量。

```
pub fn fork(
        &self,
        mapper: MapperRef,
        alloc: FrameAllocatorRef,
        stack_offset_count: u64, // Number of existing children, used to offset
the new stack
    ) -> Self {
        // 1. Calculate new stack range for the child
        let new_stack_top_addr = STACK_MAX - (stack_offset_count *
STACK_MAX_SIZE) - (self.usage * Size4KiB::SIZE);

        if new_stack_top_addr < STACK_MAX_SIZE { // Arbitrary lower bound to
prevent issues
            panic!("Out of stack space for new process");
        }

        let new_stack_top_page =
Page::<Size4KiB>::containing_address(VirtAddr::new(new_stack_top_addr));
        let new_stack_start_page = new_stack_top_page - self.usage + 1;
        let new_stack_range = Page::range(new_stack_start_page,
new_stack_top_page + 1);

        // 2. Allocate and map new stack for child

        let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE |
PageTableFlags::USER_ACCESSIBLE;
        for page in new_stack_range.clone() { // Iterate over a clone as
new_stack_range is used later
            let frame = alloc
                .allocate_frame()
                .ok_or(MapToError::<Size4KiB>::FrameAllocationFailed)
                .expect("Stack fork: Frame allocation failed for child stack");
            unsafe {
                mapper
                    .map_to(page, frame, flags, alloc)
                    .expect("Stack fork: Failed to map child stack page")
                    .flush();
            }
        }
```

```
        // 3. Copy the *entire stack* from parent to child
        let parent_stack_bottom_addr = self.range.start.start_address().as_u64();
        let child_stack_bottom_addr =
new_stack_range.start.start_address().as_u64();
        Self::clone_range(parent_stack_bottom_addr, child_stack_bottom_addr,
self.usage);
        // 4. Return the new stack
        Self {
            range: new_stack_range,
            usage: self.usage, // Child stack initially has the same usage as
parent
        }
    }
```

注意： 注意: 文档中提到 这样的栈布局在复杂情况下可能会造成栈复用，在这种情况下进行 map_range 会失败，从而你可以继续寻找合适的偏移。我当前实现的版本没有包含这种循环尝试的逻辑，这是一个可以后续优化的地方。

2.2. 接下来实现 `ProcessVm::fork`，首先在 `paging.rs` 中实现共享页表的 `fork` 方法，并添加 `using_count` 和更新 `Debug` 实现，以记录父子进程共用的页表。

```
impl Debug for PageTableContext {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        // ...
            .field("refs", &self.using_count())
        // ...
    }
}
/// Returns the number of strong references to this page table context.
    pub fn using_count(&self) -> usize {
        Arc::strong_count(&self.reg)
    }
    /// Creates a new `PageTableContext` that shares ownership of the underlying
    /// page table (CR3 register value). This is used for `fork`.
    pub fn fork(&self) -> Self {
        // Forked process shares the page table by cloning the Arc.
        Self {
            reg: self.reg.clone(),
        }
    }
```

2.3. 现在 `PageTableContext` 已经准备好了，我可以在 `pkg/kernel/src/proc/process.rs` 中实现 `ProcessInner::fork` 和 `Process::fork` 方法了，它们是 `fork` 逻辑的核心。

- 对于 `ProcessInner::fork`：
  - `fork the process virtual memory struct`：
    调用 `self.proc_vm.as_ref().unwrap().fork(stack_offset_count)`。这里的 `stack_offset_count` 需要确定。一个简单的方法是使用父进程当前的 `children.len()` 作为偏移计数。
  - `calculate the real stack offset`:这一步与上一步的 `stack_offset_count` 相关，并且涉及到子进程上下文 `rsp` 的调整。父进程和子进程虽然共享相同的代码和数据（通过共享页表，未来可能是 `COW`），但它们的栈是独立且内容在 `fork` 时刻是相同的。子进程的 `rsp` 应该相对于其新栈的栈顶，并且保持与父进程 `rsp` 在其栈中的相对位置一致。当 `Stack::fork` 复制栈内容后，子进程的 `rsp` 应该与父进程的 `rsp` 具有相同的偏移量，但是是相对于子进程自己的栈。由于我们完整复制了栈内容，并且假设栈指针 `rsp` 存储在 `ProcessContext` 中，那么子进程的 `context.rsp` 应该与父进程的 `context.rsp` 相同（因为它们指向复制后的栈中的相同相对位置）。然而，文档中提到 **“update rsp in interrupt stack frame”**。这通常意味着子进程的内核栈帧中的 `rsp` 需要被设置为子进程用户栈的某个值。在 `fork` 之后，子进程从内核态返回用户态时，会从这个内核栈帧中恢复寄存器。父进程的 `context` 是在系统调用发生时保存的，子进程会得到这个 `context` 的一个副本。如果 `ProcessContext` 直接包含了用户态的 `rsp`，那么简单复制 `context` 后，子进程的 `rsp` 就会指向其新栈的正确位置。如果 `rsp` 是指内核栈指针，那么这里可能需要更复杂的处理，或者是指 `ProcessContext` 结构中代表用户 `rsp` 的字段。鉴于 `Stack::fork` 复制了整个栈，并且 `ProcessVm::fork` 创建了新的 `Stack` 对象，子进程的 `ProcessContext`（从父进程复制而来）中的 `rsp` 应该已经是正确的（指向其新用户栈的相应位置）。我将假设 `context.rsp` 是用户栈指针，并且在复制 `ProcessContext` 后其值对于子进程是有效的。`"update rsp in interrupt stack frame" 这一条可能更多的是指确保 context` 被正确复制和使用。
  - `set the return value 0 for child with context.set_rax`:子进程的 `fork` 返回值是 0，这需要修改子进程的 `ProcessContent` 副本的 `rax` 寄存器。
  - `clone the process data struct`: `self.proc_data.clone()`。`ProcessData` 需要实现 `Clone`，不过 `pkg/kernel/src/proc/data.rs` 中的 `ProcessData` 结构体已经 `#[derive(Debug, Clone)]`，所以它已经是 `Clone`-able 的。
  - `construct the child process inner`：创建一个新的 `ProcessInner` 实例，包含新的 `proc_vm`，克隆的 `proc_data`，更新后的 `context`，父进程的 `week` 引用，空的 `children` 列表等。

```rust
pub fn fork(&mut self, parent_weak_ref: Weak<Process>, parent_pid: ProcessId) -> ProcessInner {
        // FIXME: fork the process virtual memory struct
        // Use parent's current children count to determine stack offset for the new child.
        // This assumes children are added to parent's list *after* this call.
        let stack_offset_count = self.children.len() as u64;
        let child_vm = self
            .proc_vm
            .as_ref()
            .expect("Parent ProcessVm is None during fork")
            .fork(stack_offset_count);

        // FIXME: calculate the real stack offset - This is handled by Stack::fork and ProcessVm::fork
        // FIXME: update `rsp` in interrupt stack frame - Parent's context is cloned, rsp should be correct relative to the new stack

        // Clone the parent's context for the child.
        let mut child_context = self.context.clone();
        // FIXME: set the return value 0 for child with `context.set_rax`
        child_context.set_rax(0); // Child's fork returns 0

        // FIXME: clone the process data struct

        // ProcessData already derives Clone. If it contains Arcs, they will be shared.

        let child_proc_data = self
            .proc_data
            .as_ref()
            .expect("Parent ProcessData is None during fork")
            .clone();

        // FIXME: construct the child process inner
        // Child gets a new name (e.g., "parent_name-child") or inherits parent's name.
        // For simplicity, let's append "-child" to parent's name.
        // A more robust solution might involve a counter or a different naming scheme.
        let child_name = alloc::format!("{}-child-{}", self.name, parent_pid.0); // Use parent_pid to make it somewhat unique initially
```

```
        debug!(
            "ProcessInner::fork: Parent name: {}, Parent PID: {}, Child name:
{}",
            self.name, parent_pid, child_name
        );

        ProcessInner {
            name: child_name,
            parent: Some(parent_weak_ref),
            children: Vec::new(), // Child starts with no children
            ticks_passed: 0,      // Child starts with 0 ticks
            status: ProgramStatus::Ready, // Child is ready to run
            context: child_context,
            exit_code: None,
            proc_data: Some(child_proc_data),
            proc_vm: Some(child_vm),
        }
        // NOTE: return inner because there's no pid record in inner
    }
```

- 对于 `Process::fork`：
  - `lock inner as write`：获取父进程 `self.inner.write()`。
  - `inner fork with parent week ref`：调用 `inner,fork(Arc::downgrade(self))`。
  - `make the arc of child`：使用 `Process::new_with_pid`（或 `Process::new`）创建一个新的 `Arc<process>` 包裹子进程的 `ProcessInner`。子进程需要一个新的 `ProcessId`。
  - `add child to current process's children list`：将子进程的 `Arc<Process>` 添加到父进程的 `children` 向量中。
  - `set fork ret value for parent with context.set_rax`：父进程的 `fork` 返回值是子进程的 `PID`。这需要修改父进程 `ProcessContext` 中的 `rax`。
  - `mark the child as ready & return it`：子进程创建后状态应为 `Ready`。返回子进程的 `Arc<Process>`。

```
pub fn fork(self: &Arc<Self>) -> Arc<Self> {
        // FIXME: lock inner as write
        let mut parent_inner = self.write();

        // FIXME: inner fork with parent weak ref
        // The number of existing children is used as the stack_offset_count
        let child_inner_template = parent_inner.fork(Arc::downgrade(self),
self.pid);
```

```
        // FOR DBG: maybe print the child process info
        //          e.g. parent, name, pid, etc.
        let child_pid = ProcessId::new(); // Assign a new PID for the child
        trace!(
            "Parent {}#{} forking. Child will be {}#{}",
            parent_inner.name(),
            self.pid(),
            child_inner_template.name, // Name is cloned in ProcessInner::fork
            child_pid
        );

        // FIXME: make the arc of child
        // Construct the child Process struct with a new PID
        let child_proc = Arc::new(Process {
            pid: child_pid,
            inner: Arc::new(RwLock::new(child_inner_template)),
        });

        // FIXME: add child to current process's children list
        parent_inner.children.push(child_proc.clone());

        // FIXME: set fork ret value for parent with `context.set_rax`
        // Parent's fork returns the child's PID
        parent_inner.context.set_rax(child_pid.0 as usize);
        // FIXME: mark the child as ready & return it
        // Child is already marked as Ready in ProcessInner::fork
        // The child_proc Arc is returned
        child_proc
    }
```

2.4. 至此，`Processs`和`ProcessInner`相关的`fork`逻辑已经完成。接下来我们实现`ProcessManager::fork(&self)`方法，它位于`pkg/kernel/src/proc/manager.rs`。

- 实现思路：
  - `get current process`：使用`self.current()`获取当前正在运行的进程。
  - `fork to get child`：调用当前进程的`Process::fork()`方法，这将返回一个`Arc<Process>`指向新创建的子进程。
  - `add child to process list`：使用`self.add_proc(child.pid(), child.clone())`将子进程添加到`ProcessManager`的全局进程列表中。
  - （可选调试）打印就绪队列。

这个实现比较直接，主要是调用我们之前已经完成的方法。

```rust
pub fn fork(&self) {
        // FIXME: get current process
        let parent_process = self.current();
        trace!(
            "ProcessManager: Forking current process {}#{}",
            parent_process.read().name(),
            parent_process.pid()
        );

        // FIXME: fork to get child
        // This will internally set the parent's rax to child_pid
        // and child's rax to 0.
        let child_process = parent_process.fork(); // Process::fork()
        trace!(
            "ProcessManager: Forked. Child PID: {}, Parent PID: {}",
            child_process.pid(),
            parent_process.pid()
        );

        // FIXME: add child to process list
        self.add_proc(child_process.pid(), child_process.clone());

        // Child process is already marked as Ready by Process::fork →
ProcessInner::fork.
        // It will be added to the ready queue later by the top-level syscall
handler.

        // FOR DBG: maybe print the process ready queue?
        debug!("ProcessManager: Fork complete. Current ready queue: {:?}",
self.ready_queue.lock());
    }
```

调试信息:

添加了 `trace!` 和 `debug!` 日志，用于在 `fork` 过程中输出父子进程的信息以及就绪队列的状态，方便调试。

> **注意：** 关于就绪队列的说明:
> 在 ProcessManager::fork 中, 子进程被创建并加入到全局进程列表, 但没有直接将其 PID 加入到 ready_queue。
> 这是因为文档中顶层 fork 函数的 FIXME 包含了 FIXME: push to child & parent to ready queue。这表明将父子进程加入就绪队列的操作应该在更高层（系统调用处理函数）完成。子进程在 ProcessInner::fork 中已经被设置为 ProgramStatus::Ready。

2.5. 现在，所有较低层级的 `fork` 实现（`Stack::fork`，`ProcessVm::fork`，`PageTableContext::fork`，`ProcessInner::fork`，`Process::fork`，`ProcessManager::fork`）都已经完成。最后一步是实现顶层的 `fork` 系统调用处理函数。

- 创建`sys_fork`函数:

```
pub fn sys_fork(context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        manager.save_current(context); // FIXME: save_current as parent
        let child_pid = manager.fork(); // FIXME: fork to get child
        manager.push_ready(child_pid); // FIXME: push to child & parent to ready
queue (child part)
        let _next_pid = manager.switch_next(context); // FIXME: switch to next
process
    });
}
```

这部分实现较为直接，都是调用的已经实现的方法。

- `dispatcher` 在 `Syscall::Fork` 分支调用 `sys_fork()`。

```
Syscall::Fork => {
    sys_fork(context);
},
```

2.6. 至此，我们完成了`fork`的实现:

- `Stack (pkg/kernel/src/proc/vm/stack.rs)`: 实现了 `Stack::fork()`。
- `PageTableContext (pkg/kernel/src/proc/paging.rs)`: 实现了页表共享的 `fork()` 和 `using_count()`。
- `ProcessVm (pkg/kernel/src/proc/vm/mod.rs)`: 实现了 `ProcessVm::fork()`。
- `Process/ProcessInner (pkg/kernel/src/proc/process.rs)`: 实现了 `ProcessInner::fork()` 和 `Process::fork()`。
- `ProcessManager (pkg/kernel/src/proc/manager.rs)`: 实现了`ProcessManager::fork()`。
- `Processor Module (pkg/kernel/src/proc/mod.rs)`: `processor` 模块已设为公共（`pub mod processor;`），以解决之前的模块私有错误（尽管在当前 `sys_fork` 的最终版本中，`processor` 模块未被直接使用，但保持其公共性可能对项目其他部分有益或符合预期结构）。
- `Syscall Dispatcher (pkg/kernel/src/interrupt/syscall/mod.rs)`: 包含一个简洁的 `sys_fork` 函数，严格按照文档中的 `FIXME` 步骤实现 `fork` 逻辑。

3. 功能测试

在进行测试之前，我发现我忘记正在`syscall.rs`中添加`sys_fork`的定义了，导致了`error`。

```
#[inline(always)]
pub fn sys_fork() -> u16 {
    syscall!(Syscall::Fork) as u16
}
```

接下来我们开始进行测试，首先在`pkg/app`下创建`fork_test`应用，然后重新编译运行整个项目文件，却发现得到的输出结果不对，如下：

```
ysos> ls
可用的应用程序列表:
[+] App list: factorial, hello, shell, counter, fork_test
ysos> run fork_test
正在运行程序: fork_test
进程ID: 4
I am the child process
child read value of M: 0xdeadbeef
child changed the value of M: 0x2333
```

显然是不完整的，根据`main.rs`的实现，父进程应该在子进程结束后打印一系列信息。经过排查，我发现在实现`fork`的过程中有下面几个问题：

1. 父进程调度：

`ProcessManager::save_current (pkg/kernel/src/proc/manager.rs:80)`只有在进程状态已经是`ProgramStatus::Ready`时才会将其加入就绪队列。而`fork`调用时，父进程状态是`Running`，所以它不会被`save_current`自动入队，导致`fork`后父进程无法再次被调度执行。

2. 返回值：`fork`系统调用需要为父进程返回子进程的`PID`，为子进程返回`0`。这需要在它们各自的上下文中正确设置`rax`寄存器。

修改如下：

- 修改了`sys_fork`的逻辑：
  - 在`manager.save_current(context)`保存父进程上下文之后，并且在`manager.fork()`创建了子进程之后，我们显式地获取了父进程对象，将其状态`status`修改为`ProgramStatus::Ready`，然后调用`manager.push_ready(parent_pid)`将父进程加入就绪队列。
- 同时，确保了：
  - 子进程的上下文（在`ProcessInner::fork (pkg/kernel/src/proc/process.rs:292)`中创建和修改）的`rax`被设置为 0。

- 父进程被保存的上下文（在 `Process::fork (pkg/kernel/src/proc/process.rs:159)` 中修改）的 `rax` 被设置为子进程的 `PID`。
- 父进程的当前活动上下文（在 sys_fork (pkg/kernel/src/interrupt/syscall/mod.rs:61) 中）的 `rax` 也被设置为子进程的 `PID`，作为系统调用的直接返回值。

改完之后我发现运行时会引发 Page Fault，于是继续排查，最终发现问题所在：

尽管 `Stack::fork (pkg/kernel/src/proc/vm/stack.rs)` 为子进程创建了一个新的、独立的栈空间并复制了父进程的栈内容，但在 `ProcessInner::fork` 中，子进程继承的上下文中，其 `stack_frame.stack_pointer (RSP)` 仍然是父进程的旧 `RSP`。这导致子进程在自己的新栈上进行操作时，`RSP` 指向错误的位置，极易引发栈错误和 `Page Fault`。

解决方案：

1. 为了解决因字段私有性导致的访问问题，我们首先在 `Stack` 结构体 (pkg/kernel/src/proc/vm/stack.rs:74) 中添加了公共方法 `start_address()`。
2. 接着，在 `ProcessVm` 结构体 `(pkg/kernel/src/proc/vm/mod.rs:211)` 中添加了公共方法 `stack_start_address()`。
3. 然后，在 `ProcessContext` 结构体 `(pkg/kernel/src/proc/context.rs:36)` 中添加了公共方法 `set_rsp()`。
4. 最后，在 `ProcessInner::fork() (pkg/kernel/src/proc/process.rs:302)` 中，我们利用这些新添加的 `getter` 和 `setter` 方法：
   - 获取父进程当前 `RSP (parent_rsp_virtaddr)`。
   - 获取父进程栈的起始虚拟地址 `(parent_stack_start_virtaddr)`。
   - 计算父进程 `RSP` 相对于其栈起始地址的偏移量 `(rsp_offset_from_stack_start)`。
   - 获取子进程新栈的起始虚拟地址 `(child_stack_start_virtaddr)`。
   - 子进程的新 `RSP` 计算为 `child_stack_start_virtaddr` + `rsp_offset_from_stack_start`。
   - 使用 `child_context.set_rsp()` 将计算出的新 `RSP` 更新到子进程的上下文中。

> **注意：** 这里包括 sys::fork 和 stack::fork 以及 ProcessInner::fork 都做了较大改动，完整的新版代码放在报告最后。

重新编译运行得到了正确的结果（如下图所示）： 分析一下输出：

- 程序运行开始时打印：`正在运行程序: fork_test`
- 父进程的进程 ID 为 4：`进程ID: 4`（这是父进程 `fork_test` 的 PID）
- 子进程（PID 5）打印：`I am the child process`
- 父进程（PID 4）打印：`I am the parent process`
- `sys_stat()` 的输出显示了系统当前状态：
  - 内核进程（PID 1）
  - `shell` 进程（PID 2）
  - 父进程 `fork_test`（PID 4），状态为 `Running`

- 子进程 `fork_test-child-4`（PID 5），状态为 `Ready`（符合预期，父进程在运行，子进程在等待被调度）
- 就绪队列状态：`Queue : [5, 2, 1]`（子进程在队列头部，符合预期）

· 父进程打印：`Waiting for child to exit...`

· 子进程访问共享数据段：

- 读取值：`child read value of M: 0xdeadbeef`
- 修改值：`child changed the value of M: 0x2333`

· 子进程退出，父进程打印：`Child exited with status 64`（子进程退出状态正确）

· 父进程读取共享数据：`parent read value of M: 0x2333`（成功读取到子进程修改的值，因为数据段是共享的，符合实验文档）

· 最后程序退出：`程序 'fork_test' 已退出，返回值: 1056`（父进程正常退出，返回值也正确）

```
ysos> ls
可用的应用程序列表:
[+] App list: factorial, hello, shell, counter, fork_test
ysos> run fork_test
正在运行程序: fork_test
进程ID: 4
I am the child process
I am the parent process
  PID | PPID | Process Name |  Ticks  | Status
 #  1 | #  0 | kernel       | 19859   | 0         |      0 B | Ready
 #  2 | #  1 | shell        | 19852   | 7         |     28 KiB | Ready
 #  4 | #  2 | fork_test    | 1       | 6         |     24 KiB | Running
 #  5 | #  4 | fork_test-child-4 | 1       | 6         |     24 KiB | Ready
Queue  : [5, 2, 1]
CPUs   : [0: 4]
Waiting for child to exit...
child read value of M: 0xdeadbeef
child changed the value of M: 0x2333
Child exited with status 64
parent read value of M: 0x2333
程序 'fork_test' 已退出，返回值: 1056
ysos>
```

二. 进程的阻塞与唤醒

1. 等待队列

在 `ProcessManager` 中添加等待队列字段**并为其添加初始值**。

```rust
pub struct ProcessManager {
    // ...
    wait_queue: Mutex<BTreeMap<ProcessId, BTreeSet<ProcessId>>>,
}
```

```rust
impl ProcessManager {
    pub fn new(init: Arc<Process>) -> Self {
        // ...
        Self {
            // ...
            wait_queue: Mutex::new(BTreeMap::new()),
        }
    }
```

2. 阻塞进程

2.1. 把 `block` 方法添加到 `ProcessInner` 的实现中，放在 `resume` 方法的后面。这个方法会将进程的状态设置为阻塞状态。

```rust
pub fn block(&mut self) {
    self.status = ProcessStatus::Blocked;
}
```

2.2. 为 `ProcessManager` 添加 `block` 函数，用于调用指定 `PID` 进程的 `block` 方法，并为 `ProcessManager` 添加 `wait_pid` 函数,用于将当前进程加入到目标进程的等待队列中。

- 我们需要在这里调用我们刚刚在 `ProcessInner` 中添加的 `block` 方法。由于 `self.get_proc(&pid)` 返回的是 `Option<Arc<Process>>`，我们需要通过 `proc.write().block()` 来调用。

```rust
/// Block the process with the given pid
pub fn block(&self, pid: ProcessId) {
    if let Some(proc) = self.get_proc(&pid) {
        // FIXME: set the process as blocked
    }
}
```

- 我们需要使用 `BTreeMap` 的 `entry` API。`wait_queue` 是一个 `Mutex<BTreeMap<ProcessId, BTreeSet<ProcessId>>>`。键是被等待的进程 `pid`，值是一个 `BTreeSet`，包含所有等待这个 `pid` 的进程的 `ProcessId`。我们需要获取当前正在执行的进程的 `ProcessId`（通过 `processor::get_pid()`），然后将其添加到对应 `pid` 的 `BTreeSet` 中。

```rust
pub fn wait_pid(&self, pid: ProcessId) {
        let mut wait_queue = self.wait_queue.lock();
        let current_pid = processor::get_pid();
        wait_queue.entry(pid).or_default().insert(current_pid);
    }
```

2.3. 在 `pkg/kernel/src/proc/mod.rs` 中，修改 `wait_pid` 系统调用的实现，添加 `ProcessContext` 参数来确保可以进行可能的切换上下文操作（意味着当前进程被阻塞，需要切换到下一个进程）。（代码已给，略）

3. 唤醒进程

3.1. 为 `ProcessManager` 添加 `wake_up` 函数,此函数负责设置被唤醒进程的返回状态、将其状态置为 `Ready`，并推入就绪队列。

```rust
/// Wake up the process with the given pid
    ///
    /// If `ret` is `Some`, set the return value of the process
    pub fn wake_up(&self, pid: ProcessId, ret: Option<isize>) {
        if let Some(proc) = self.get_proc(&pid) {
            let mut inner = proc.write();
            // Only attempt to wake up if the process is actually Blocked.
            if inner.status() == ProgramStatus::Blocked {
                if let Some(ret_val) = ret {
                    // Set the return value (e.g., for wait_pid)
                    inner.context.set_rax(ret_val as usize);
                }
                // Set the process status to Ready
                inner.pause(); // .pause() sets status to ProgramStatus::Ready

                // Drop the lock before pushing to ready_queue
                drop(inner);

                // Push the process to the ready queue
                self.push_ready(pid);
            }
        }
    }
```

3.2. 修改 `ProcessManager` 中的 `kill` 函数,当一个进程被杀死时，此函数现在会检查并唤醒所有在该进程的等待队列中的其它进程，并将被杀死进程的退出码作为唤醒进程的返回值。

```rust
pub fn kill(pid: ProcessId, ret: isize) {
    // ...

    if let Some(pids) = self.wait_queue.lock().remove(&pid) {
        for pid in pids {
            self.wake_up(pid, Some(ret));
        }
    }
}
```

4. 阶段性成果

- 尝试在你的 Shell 中启动另一个 Shell，然后在其中利用 ps 打印进程信息：

  前一个 Shell 应当被阻塞 (Blocked)，直到后一个 Shell 退出。

- 如图所示, 完全符合预期, 本来第一个 `shell` 是处于 `running` 状态, 在其中执行 `run shell` 之后再使用 `ps` 查看进程列表的时候可以看到第一个 shell 的状态已经由 `running` 变为 `ready`，而当我们使用 `exit` 退出第二个 `shell` 之后，可以看到原来的也就是第一个 shell 的状态由 `ready` 恢复到了 `running`, 说明我们到已经完全实现了“进程的阻塞与唤醒”部分!

```
[+] App list: factorial, hello, shell, counter, fork_test
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name |  Ticks   | Status
 #  1 | #  0 | kernel       | 7268     | 0          |      0 B | Ready
 #  2 | #  1 | shell        | 7260     | 7          |     28 KiB | Running
Queue  : [1]
CPUs   : [0: 2]
ysos> ls
可用的应用程序列表:
[+] App list: factorial, hello, shell, counter, fork_test
ysos> run shell
正在运行程序: shell
进程ID: 3
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name |  Ticks   | Status
 #  1 | #  0 | kernel       | 24884    | 0          |      0 B | Ready
 #  2 | #  1 | shell        | 24876    | 7          |     28 KiB | Ready
 #  3 | #  2 | shell        | 9027     | 7          |     28 KiB | Running
Queue  : [2, 1]
CPUs   : [0: 3]
ysos> exit
退出Shell...
程序 'shell' 已退出，返回值: 0
ysos> ps
当前运行的进程列表:
  PID | PPID | Process Name |  Ticks   | Status
 #  1 | #  0 | kernel       | 133471   | 0          |      0 B | Ready
 #  2 | #  1 | shell        | 133463   | 7          |     28 KiB | Running
Queue  : [1]
CPUs   : [0: 2]
ysos>
```

三. 并发与锁机制

1. 在初步了解（回顾）完实验指导书给出的原子指令、临界区问题后，我们尝试实现简单的锁机制——自旋锁 `SpinLock`，代码位于 `pkg/lib/src/sync.rs`：

- **实现思路：**
  1. 对于 `acquire` 方法，使用 `compare_exchange` 原子操作。如果锁是可用的（`false`），就将其设置为 `true`。如果锁不可用，就使用 `spin_loop` 进行自旋等待。
  2. 对于 `release` 方法，我将使用 `store` 原子操作将锁的状态设置回 `false`。

```
    pub fn acquire(&self) {
        while self
            .bolt
            .compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed)
            .is_err()
        {
            // 如果锁被占用，则自旋
            while self.bolt.load(Ordering::Relaxed) {
                spin_loop();
            }
        }
    }

    pub fn release(&self) {
        self.bolt.store(false, Ordering::Release);
    }
```

2. 在回顾完指导书中给出的信号量相关内容后首先根据 `FIXME` 的注释内容来完成 `Semaphore` 和 `SemaphoreSet` 的方法。

```
    pub fn wait(&mut self, pid: ProcessId) -> SemaphoreResult {
        if self.count == 0 {
            self.wait_queue.push_back(pid);
            SemaphoreResult::Block(pid)
        } else {
            self.count -= 1;
            SemaphoreResult::Ok
        }
    }

    pub fn signal(&mut self) -> SemaphoreResult {
        if let Some(pid) = self.wait_queue.pop_front() {
            SemaphoreResult::WakeUp(pid)
        } else {
            self.count += 1;
            SemaphoreResult::Ok
        }
    }
}
```

```rust
impl SemaphoreSet {
    pub fn insert(&mut self, key: u32, value: usize) -> bool {
        trace!("Sem Insert: <{:#x}>{}", key, value);
        self.sems
            .insert(SemaphoreId::new(key), Mutex::new(Semaphore::new(value)))
            .is_none()
    }

    pub fn remove(&mut self, key: u32) -> bool {
        trace!("Sem Remove: <{:#x}>", key);
        self.sems.remove(&SemaphoreId::new(key)).is_some()
    }

    /// Wait the semaphore (acquire/down/proberen)
    pub fn wait(&self, key: u32, pid: ProcessId) -> SemaphoreResult {
        let sid = SemaphoreId::new(key);
        if let Some(sem_mutex) = self.sems.get(&sid) {
            sem_mutex.lock().wait(pid)
        } else {
            SemaphoreResult::NotExist
        }
    }

    /// Signal the semaphore (release/up/verhogen)
    pub fn signal(&self, key: u32) -> SemaphoreResult {
        let sid = SemaphoreId::new(key);
        if let Some(sem_mutex) = self.sems.get(&sid) {
            sem_mutex.lock().signal()
        } else {
            SemaphoreResult::NotExist
        }
    }
}
```

3. 接下来实现信号量需要的四个操作 `new`，`remove`，`signal`，`wait`，以及它们在内核态和用户态的接口形式。

1. 内核态实现
   - 补充 `sem_wait` 中进程阻塞的逻辑：保存上下文、阻塞当前进程、切换到下一进程。
   - 补充 `sem_signal` 中进程唤醒的逻辑：调用进程管理器的 `wake_up` 方法。
   - 实现 `new_sem` 和 `remove_sem` 函数，用于创建和删除信号量。
   - 完善 `sys_sem` 系统调用入口，根据操作码分发到具体的信号量操作函数。

所有操作均考虑了中断屏蔽以保证原子性。

```
use crate::process::{get_process_manager, ProcessContext};
use crate::processor;
use crate::syscall::SyscallArgs;
use crate::x86_64;

pub fn new_sem(key: u32, value: usize) -> usize {
    let manager = get_process_manager();
    if manager.current().write().sem_new(key, value) {
        0
    } else {
        1
    }
}

pub fn remove_sem(key: u32) -> usize {
    let manager = get_process_manager();
    if manager.current().write().sem_remove(key) {
        0
    } else {
        1
    }
}

pub fn sem_signal(key: u32, context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        // let pid = processor::get_pid(); // Not directly used in signal logic
itself, but good for consistency if needed later
        let ret = manager.current().write().sem_signal(key);
        match ret {
            SemaphoreResult::Ok => context.set_rax(0),
            SemaphoreResult::NotExist => context.set_rax(1), // Using 1 for
NotExist as per convention
            SemaphoreResult::WakeUp(pid_wake) => {
                manager.wake_up(pid_wake, None);
                context.set_rax(0);
            }
            _ => unreachable!("sem_signal should not block"),
        }
    })
}
```

```rust
pub fn sem_wait(key: u32, context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        let pid = processor::get_pid();
        let ret = manager.current().write().sem_wait(key, pid);
        match ret {
            SemaphoreResult::Ok => context.set_rax(0),
            SemaphoreResult::NotExist => context.set_rax(1), // Using 1 for
NotExist
            SemaphoreResult::Block(pid_block) => {
                assert_eq!(pid_block, pid, "SemaphoreResult::Block should carry
the current PID");
                manager.save_current(context); // Save current process's context
                manager.block_current(None); // Block the current process

                manager.switch_next(context); // Switch to the next available
process
            }
            _ => unreachable!("sem_wait should not wake up another process
directly"),
        }
    })
}

pub fn sys_sem(args: &SyscallArgs, context: &mut ProcessContext) {
    match args.arg0 {
        0 => context.set_rax(new_sem(args.arg1 as u32, args.arg2)), // op 0:
new_sem
        1 => context.set_rax(remove_sem(args.arg1 as u32)),      // op 1:
remove_sem
        2 => sem_signal(args.arg1 as u32, context),              // op 2:
sem_signal
        3 => sem_wait(args.arg1 as u32, context),                // op 3:
sem_wait
        _ => context.set_rax(usize::MAX), // Invalid operation, return a distinct
error code
    }
}
```

2. 用户态库实现

- 为 `Semaphore` 结构体添加 `key` 字段。
- 为 `Semaphore` 实现 `init`、`remove`、`signal` 和 `wait` 方法。

这些方法通过定义相应的 `sys_xxx_sem` 内联辅助函数，使用 syscall! 宏和正确的操作码（0 for `new`，1 for `remove`，2 for `signal`，3 for `wait`）来调用内核服务。

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq, PartialOrd, Ord)]
pub struct Semaphore {
    key: u32,
}

impl Semaphore {
    pub const fn new(key: u32) -> Self {
        Self { key }
    }

    #[inline(always)]
    pub fn init(&self, value: usize) -> bool {
        sys_new_sem(self.key, value)
    }

    #[inline(always)]
    pub fn remove(&self) -> bool {
        sys_remove_sem(self.key)
    }

    #[inline(always)]
    pub fn signal(&self) -> bool {
        sys_signal_sem(self.key)
    }

    #[inline(always)]
    pub fn wait(&self) -> bool {
        sys_wait_sem(self.key)
    }
}

#[inline(always)]
fn sys_new_sem(key: u32, value: usize) -> bool {
    syscall!(Syscall::Sem, 0, key as usize, value) == 0
}

#[inline(always)]
fn sys_remove_sem(key: u32) -> bool {
    syscall!(Syscall::Sem, 1, key as usize, 0) == 0 // value is not used for
remove
}
```

```rust
#[inline(always)]
fn sys_signal_sem(key: u32) -> bool {
    syscall!(Syscall::Sem, 2, key as usize, 0) == 0 // value is not used for
signal
}

#[inline(always)]
fn sys_wait_sem(key: u32) -> bool {
    syscall!(Syscall::Sem, 3, key as usize, 0) == 0 // value is not used for wait
}
```

四. 测试任务

1. 多线程计数器

这里突然发现忘记在 `ProcessData` 中添加 `SemaphoreSet` 字段，在 `data.rs` 中：

```rust
use alloc::{collections::BTreeMap, sync::Arc};
use spin::RwLock;
use super::sync::SemaphoreSet;

use x86_64::structures::paging::{
    page::{PageRange, PageRangeInclusive},
    Page,
```

12 hidden lines

```rust
    pub(super) code_bytes: u64, // Bytes used by code/data segr
    pub(super) stack_pages: u64, // Pages used by stack
    pub(super) total_pages: u64, // Total pages used (code + st
    pub(super) semaphores: Arc<RwLock<SemaphoreSet>>,
}

impl Default for ProcessData {
```

4 hidden lines

```rust
                code_bytes: 0,
                stack_pages: 0,
                total_pages: 0,
                semaphores: Arc::new(RwLock::new(SemaphoreSet::defa
            }
        }
}
```

44 hidden lines

由于信号量的实现是进程同步的关键机制，同时信号量的操作通常是原子操作，需要在内核态下执行，以保证并发环境下的正确性和安全性，因此我把`new`、`remove`、`signal`、`wait`操作的实现放在`ProcessInner`中去。

```
pub fn sem_new(&mut self, key: u32, value: usize) -> bool {
        if let Some(pd) = self.proc_data.as_mut() {
             pd.semaphores.write().insert(key, value)
        } else {
            error!("sem_new called on ProcessInner with no proc_data");
            false
        }
    }

    pub fn sem_remove(&mut self, key: u32) -> bool {
        if let Some(pd) = self.proc_data.as_mut() {
            pd.semaphores.write().remove(key)
        } else {
            error!("sem_remove called on ProcessInner with no proc_data");
            false
        }
    }

    pub fn sem_signal(&self, key: u32) -> sync::SemaphoreResult {
        // Deref gives &ProcessData.
        if let Some(pd) = self.proc_data.as_ref() {
            pd.semaphores.read().signal(key)
        } else {
            error!("sem_signal called on ProcessInner with no proc_data");
            sync::SemaphoreResult::NotExist // Or a more generic error
        }
    }

    pub fn sem_wait(&self, key: u32, pid_to_block: ProcessId) ->
sync::SemaphoreResult {
        if let Some(pd) = self.proc_data.as_ref() {
            pd.semaphores.read().wait(key, pid_to_block)
        } else {
            error!("sem_wait called on ProcessInner with no proc_data");
            sync::SemaphoreResult::NotExist // Or a more generic error
        }
    }
```

测试程序实现思路：

将测试逻辑拆分为`test_spin()`和`test_semaphore()`两个独立的函数。

`main` 函数被修改为依次调用这两个测试函数(当前 `test_semaphore` 在 `main` 中被注释,但已实现)。

- `test_spin()`:
  - 使用全局静态 `SpinLock`。
  - 子进程通过 `do_counter_inc_spin()` 调用受 `SpinLock` 保护的计数器递增操作。
- `test_semaphore()`:
  - 使用全局静态 `Semaphore` (通过键值 `SEM_KEY` 标识)。
  - 在测试开始时初始化信号量 (值为 1,作为互斥锁),测试结束后移除信号量。
  - 子进程通过 `do_counter_inc_sem()` 调用受 `Semaphore` 保护的计数器递增操作。

两个测试都包含 `THREAD_COUNT` 个子进程,每个子进程对计数器执行 `100` 次递增。

测试结束后会验证计数器的最终值是否符合预期 (`THREAD_COUNT` * `100`),并打印通过/失败信息。

```
fn test_spin() {
    println!("--- Testing SpinLock ---");
    unsafe { COUNTER = 0; } // 重置计数器
    let mut pids = [0u16; THREAD_COUNT];

    for i in 0..THREAD_COUNT {
        let pid = sys_fork();
        if pid == 0 {
            do_counter_inc_spin();
            sys_exit(0);
        } else {
            pids[i] = pid;
        }
    }

    let cpid = sys_get_pid();
    println!("SpinLock test process #{} holds threads: {:?}", cpid, &pids);
    sys_stat();

    for i in 0..THREAD_COUNT {
        println!("#{} waiting for #{}...", cpid, pids[i]); // 可以暂时注释掉，减少输
出
        sys_wait_pid(pids[i]);
    }
```

```
    println!("SpinLock COUNTER result: {}", unsafe { COUNTER });
    if unsafe { COUNTER } == (THREAD_COUNT * 100) as isize {
        println!("SpinLock test PASSED! 🎉");
    } else {
        println!("SpinLock test FAILED! ☹ Expected {}, got {}", THREAD_COUNT *
100, unsafe { COUNTER });
    }
}
fn main() -> isize {
    test_spin();
    // test_semaphore();
    0
}

fn do_counter_inc_sem() {
    for _ in 0..100 {
        SEMAPHORE.wait(); // P 操作
        inc_counter();    // 临界区
        SEMAPHORE.signal(); // V 操作
    }
}

fn test_semaphore() {
    println!("--- Testing Semaphore ---");
    unsafe { COUNTER = 0; } // 重置计数器

    // 初始化信号量，初始值为1，用作互斥锁
    if !SEMAPHORE.init(1) {
        println!("Failed to initialize semaphore! Skipping Semaphore test. ☹");
        return;
    }
    println!("Semaphore initialized with key {:#x} and value 1.", SEM_KEY);


    let mut pids = [0u16; THREAD_COUNT];
    for i in 0..THREAD_COUNT {
        let pid = sys_fork();
        if pid == 0 {
            do_counter_inc_sem();
            sys_exit(0);
        } else {
            pids[i] = pid;
        }
    }
```

```
    let cpid = sys_get_pid();
    println!("Semaphore test process #{} holds threads: {:?}", cpid, &pids);

    for i in 0..THREAD_COUNT {
        sys_wait_pid(pids[i]);
    }

    println!("Semaphore COUNTER result: {}", unsafe { COUNTER });
    if unsafe { COUNTER } == (THREAD_COUNT * 100) as isize {
        println!("Semaphore test PASSED! 🎉");
    } else {
        println!("Semaphore test FAILED! ☹ Expected {}, got {}", THREAD_COUNT *
100, unsafe { COUNTER });
    }

    // 移除信号量
    if !SEMAPHORE.remove() {
        println!("Failed to remove semaphore with key {:#x}. This might cause
issues in subsequent runs.", SEM_KEY);
    } else {
        println!("Semaphore with key {:#x} removed successfully.", SEM_KEY);
    }
}

fn do_counter_inc_spin() {
    for _ in 0..100 {
        SPIN_LOCK.acquire();
        inc_counter(); // 这是临界区
        SPIN_LOCK.release();
    }
}
```

分别测试得到下面的结果，均正确实现 `保护该临界区，使得计数器的值最终为 800` ：使用

```
[+] App list: factorial, hello, shell, counter, fork_test
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> ls
可用的应用程序列表:
[+] App list: factorial, hello, shell, counter, fork_test
ysos> run counter
正在运行程序: counter
进程ID: 3
--- Testing SpinLock ---
SpinLock test process #3 holds threads: [4, 5, 6, 7, 8, 9, 10, 11]
  PID | PPID | Process Name |  Ticks   | Status
 #  1 | #  0 | kernel       | 12484    | 0         |      0 B | Ready
 #  2 | #  1 | shell        | 12477    | 7         |     28 KiB | Ready
 #  3 | #  2 | counter      | 6        | 6         |     24 KiB | Running
 #  4 | #  3 | counter-child-3 | 11       | 6         |     24 KiB | Ready
 #  5 | #  3 | counter-child-3 | 10       | 6         |     24 KiB | Ready
 #  6 | #  3 | counter-child-3 | 9        | 6         |     24 KiB | Ready
 #  7 | #  3 | counter-child-3 | 8        | 6         |     24 KiB | Ready
 #  8 | #  3 | counter-child-3 | 7        | 6         |     24 KiB | Ready
 #  9 | #  3 | counter-child-3 | 6        | 6         |     24 KiB | Ready
 # 10 | #  3 | counter-child-3 | 5        | 6         |     24 KiB | Ready
 # 11 | #  3 | counter-child-3 | 4        | 6         |     24 KiB | Ready
Queue  : [11, 10, 9, 8, 7, 6, 5, 4, 2, 1]
CPUs   : [0: 3]
#3 waiting for #4...
#3 waiting for #5...
#3 waiting for #6...
#3 waiting for #7...
#3 waiting for #8...
#3 waiting for #9...
#3 waiting for #10...
#3 waiting for #11...
SpinLock COUNTER result: 800
SpinLock test PASSED! 🎉
程序 'counter' 已退出，返回值: 0
ysos>
```

```
[+] App list: factorial, hello, shell, counter, fork_test
欢迎使用YSOS Shell!
输入 help 获取帮助信息
ysos> run counter
正在运行程序: counter
进程ID: 3
--- Testing Semaphore ---
Semaphore initialized with key 0x1234 and value 1.
Semaphore test process #3 holds threads: [4, 5, 6, 7, 8, 9, 10, 11]
Semaphore COUNTER result: 800
Semaphore test PASSED! 🎉
Semaphore with key 0x1234 removed successfully.
程序 'counter' 已退出，返回值: 0
ysos>
```

2. 消息队列

2.1. 创建一个用户程序 pkg/app/mq，结合使用信号量，实现一个消息队列：

实验结果：

```
ysos> run mq
正在运行程序: mq
进程ID: 12
消息队列测试开始，队列容量: 1
信号量初始化完成
生产者 #1 (PID: 14) 开始生产消息
生产者 #0 (PID: 13) 开始生产消息
生产者 #2 (PID: 15) 开始生产消息
生产者 #3 (PID: 16) 开始生产消息
生产者 #4 (PID: 17) 开始生产消息
生产者 #5 (PID: 18) 开始生产消息
生产者 #1 (PID: 14) 生产消息: 100, 队列大小: 1/1, 队列是否满: true
生产者 #6 (PID: 19) 开始生产消息
所有进程创建完成，进程ID: [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
生产者 #7 (PID: 20) 开始生产消息
  PID | PPID | Process Name |  Ticks  | Status
  #  1 | #  0 | kernel       | 47011   | 0         |      0 B | Ready
  #  2 | #  1 | shell        | 47005   | 7         |     28 KiB | Ready
  # 12 | #  2 | mq           | 7       | 7         |     28 KiB | Running
  # 13 | # 12 | mq-child-12  | 2       | 7         |     28 KiB | Blocked
  # 14 | # 12 | mq-child-12  | 9       | 7         |     28 KiB | Ready
  # 15 | # 12 | mq-child-12  | 1       | 7         |     28 KiB | Blocked
  # 16 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 17 | # 12 | mq-child-12  | 1       | 7         |     28 KiB | Blocked
  # 18 | # 12 | mq-child-12  | 2       | 7         |     28 KiB | Blocked
  # 19 | # 12 | mq-child-12  | 1       | 7         |     28 KiB | Blocked
  # 20 | # 12 | mq-child-12  | 1       | 7         |     28 KiB | Ready
  # 21 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 22 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 23 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 24 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 25 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 26 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 27 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
  # 28 | # 12 | mq-child-12  | 0       | 7         |     28 KiB | Blocked
Queue  : [20, 14, 2, 1]
CPUs   : [0: 12]
消费者 #0 (PID: 21) 开始消费消息
消费者 #1 (PID: 22) 开始消费消息
```

```
消费者 #6 (PID: 27) 消费消息: 608, 队列大小: 0/1, 队列是否空: true
生产者 #7 (PID: 20) 生产消息: 708, 队列大小: 1/1, 队列是否满: true
消费者 #0 (PID: 21) 消费消息: 708, 队列大小: 0/1, 队列是否空: true
生产者 #0 (PID: 13) 生产消息: 9, 队列大小: 1/1, 队列是否满: true
消费者 #7 (PID: 28) 消费消息: 9, 队列大小: 0/1, 队列是否空: true
生产者 #1 (PID: 14) 生产消息: 109, 队列大小: 1/1, 队列是否满: true
消费者 #1 (PID: 22) 消费消息: 109, 队列大小: 0/1, 队列是否空: true
生产者 #2 (PID: 15) 生产消息: 209, 队列大小: 1/1, 队列是否满: true
消费者 #2 (PID: 23) 消费消息: 209, 队列大小: 0/1, 队列是否空: true
生产者 #3 (PID: 16) 生产消息: 309, 队列大小: 1/1, 队列是否满: true
消费者 #3 (PID: 24) 消费消息: 309, 队列大小: 0/1, 队列是否空: true
生产者 #4 (PID: 17) 生产消息: 409, 队列大小: 1/1, 队列是否满: true
消费者 #4 (PID: 25) 消费消息: 409, 队列大小: 0/1, 队列是否空: true
生产者 #5 (PID: 18) 生产消息: 509, 队列大小: 1/1, 队列是否满: true
消费者 #5 (PID: 26) 消费消息: 509, 队列大小: 0/1, 队列是否空: true
生产者 #6 (PID: 19) 生产消息: 609, 队列大小: 1/1, 队列是否满: true
消费者 #6 (PID: 27) 消费消息: 609, 队列大小: 0/1, 队列是否空: true
消费者 #0 (PID: 21) 完成消费，退出
生产者 #7 (PID: 20) 生产消息: 709, 队列大小: 1/1, 队列是否满: true
生产者 #0 (PID: 13) 完成生产，退出
消费者 #7 (PID: 28) 消费消息: 709, 队列大小: 0/1, 队列是否空: true
生产者 #1 (PID: 14) 完成生产，退出
消费者 #1 (PID: 22) 完成消费，退出
生产者 #2 (PID: 15) 完成生产，退出
消费者 #2 (PID: 23) 完成消费，退出
生产者 #3 (PID: 16) 完成生产，退出
消费者 #3 (PID: 24) 完成消费，退出
生产者 #4 (PID: 17) 完成生产，退出
消费者 #4 (PID: 25) 完成消费，退出
生产者 #5 (PID: 18) 完成生产，退出
消费者 #5 (PID: 26) 完成消费，退出
生产者 #6 (PID: 19) 完成生产，退出
消费者 #6 (PID: 27) 完成消费，退出
生产者 #7 (PID: 20) 完成生产，退出
消费者 #7 (PID: 28) 完成消费，退出
所有进程已退出，最终消息队列的消息数量: 0
测试通过！队列为空，所有消息都被消费了。
程序 'mq' 已退出，返回值: 0
ysos>
```

· 队列的元素计数是否没有超过容量?

是的，在所有测试中，消息队列的元素数量始终没有超过其设定的容量。

· 生产者和消费者是否能够交替工作?

是的。

· 容量为 1 时，严格交替。

· 容量为 4 和 8 时，表现为批次的生产和消费，交替进行。

- 容量为 16 时，由于容量较大，生产者和消费者可以更长时间地进行各自的操作，但整体上仍然是交替的，以确保消息的流动。
- 尝试观察到多个生产者或消费者同时进行同种操作的情况。

是的，尤其是在队列容量大于 1 时。我们可以看到多个生产者几乎同时“开始生产”，然后依次获取锁并向队列中添加消息，直到队列满或它们完成各自的生产任务。消费者同理。sys_stat 中多个同类进程处于 Blocked 状态也佐证了它们在竞争资源或等待条件满足。

3. 哲学家的晚饭

1. 某些哲学家能够成功就餐，即同时拿到左右两侧的筷子。

实现思路：

- `semaphore_array!` 宏来创建筷子的信号量
- `sys_fork()` 来创建哲学家进程
- `sys_get_pid()` 来获取进程 ID 作为随机数种子
- 引入 `rand` 和 `rand_chacha` 来生成随机延迟

```
哲学家 1 尝试拿起左边的筷子 1... 🥢
哲学家 0 正在就餐... 🍜
哲学家 2 正在就餐... 🍜
哲学家 1 尝试拿起右边的筷子 2... 🥢
哲学家 0 放下了筷子... ⬇️
哲学家 2 放下了筷子... ⬇️
哲学家 1 正在就餐... 🍜
哲学家 0 正在思考... 🤔
哲学家 2 正在思考... 🤔
哲学家 1 放下了筷子... ⬇️
哲学家 0 尝试拿起右边的筷子 1... 🥢
哲学家 2 尝试拿起右边的筷子 3... 🥢
哲学家 1 正在思考... 🤔
哲学家 0 尝试拿起左边的筷子 0... 🥢
哲学家 3 尝试拿起右边的筷子 4... 🥢
哲学家 1 尝试拿起左边的筷子 1... 🥢
哲学家 3 正在就餐... 🍜
哲学家 2 尝试拿起左边的筷子 2... 🥢
哲学家 0 正在就餐... 🍜
哲学家 3 放下了筷子... ⬇️
哲学家 2 正在就餐... 🍜
哲学家 1 尝试拿起右边的筷子 2... 🥢
哲学家 3 正在思考... 🤔
哲学家 0 放下了筷子... ⬇️
哲学家 3 尝试拿起左边的筷子 3... 🥢
哲学家 0 正在思考... 🤔
哲学家 2 放下了筷子... ⬇️
哲学家 0 尝试拿起右边的筷子 1... 🥢
哲学家 3 尝试拿起右边的筷子 4... 🥢
哲学家 2 正在思考... 🤔
哲学家 1 正在就餐... 🍜
哲学家 3 正在就餐... 🍜
哲学家 2 尝试拿起右边的筷子 3... 🥢
哲学家 1 放下了筷子... ⬇️
哲学家 1 正在思考... 🤔
哲学家 0 尝试拿起左边的筷子 0... 🥢
哲学家 1 尝试拿起左边的筷子 1... 🥢
哲学家 0 正在就餐... 🍜
```

```rust
#![no_std]
#![no_main]

extern crate alloc;
extern crate lib;

use lib::*;
use rand::prelude::*;
use rand_chacha::ChaCha20Rng;

// 定义常量
const PHILOSOPHER_COUNT: usize = 5;
static CHOPSTICKS: [Semaphore; PHILOSOPHER_COUNT] = semaphore_array![0, 1, 2, 3,
4];

fn think(id: usize, rng: &mut ChaCha20Rng) {
    crate::println!("哲学家 {} 正在思考... 🤔", id);
    // 随机思考时间
    for _ in 0..rng.gen_range(100000..500000) {
        // 空循环模拟延迟
    }
}

fn eat(id: usize, rng: &mut ChaCha20Rng) {
    crate::println!("哲学家 {} 正在就餐... 🍜", id);
    // 随机就餐时间
    for _ in 0..rng.gen_range(100000..300000) {
        // 空循环模拟延迟
    }
}

fn philosopher(id: usize) -> ! {
    let mut rng = ChaCha20Rng::seed_from_u64(sys_get_pid() as u64);

    // 左右筷子的编号
    let left = id;
    let right = (id + 1) % PHILOSOPHER_COUNT;

    loop {
        think(id, &mut rng);

        // 根据哲学家ID的奇偶性决定先拿哪边的筷子，避免死锁
        if id % 2 == 0 {
            // 偶数号哲学家先拿右边的筷子
            crate::println!("哲学家 {} 尝试拿起右边的筷子 {}... 🥢", id, right);
```

```
            CHOPSTICKS[right].wait();
            crate::println!("哲学家 {} 尝试拿起左边的筷子 {}... 🥢", id, left);
            CHOPSTICKS[left].wait();
        } else {
            // 奇数号哲学家先拿左边的筷子
            crate::println!("哲学家 {} 尝试拿起左边的筷子 {}... 🥢", id, left);
            CHOPSTICKS[left].wait();
            crate::println!("哲学家 {} 尝试拿起右边的筷子 {}... 🥢", id, right);
            CHOPSTICKS[right].wait();
        }

        eat(id, &mut rng);

        // 放下筷子
        CHOPSTICKS[left].signal();
        CHOPSTICKS[right].signal();
        crate::println!("哲学家 {} 放下了筷子... ⬇", id);
    }
}

fn main() -> isize {
    // 初始化信号量
    for i in 0..PHILOSOPHER_COUNT {
        if !CHOPSTICKS[i].init(1) {
            crate::println!("初始化筷子{}失败 ×", i);
            return -1;
        }
    }
    crate::println!("哲学家就餐问题开始... 🏃");
    crate::println!("共有 {} 位哲学家和 {} 双筷子 🥢", PHILOSOPHER_COUNT,
PHILOSOPHER_COUNT);

    // 创建哲学家进程
    let mut children = [0; PHILOSOPHER_COUNT-1];
    for i in 0..PHILOSOPHER_COUNT-1 {
        let pid = sys_fork();
        if pid == 0 {
            philosopher(i);
        } else {
            crate::println!("创建哲学家 {} (PID: {}) ✅", i, pid);
            children[i] = pid;
        }
    }
```

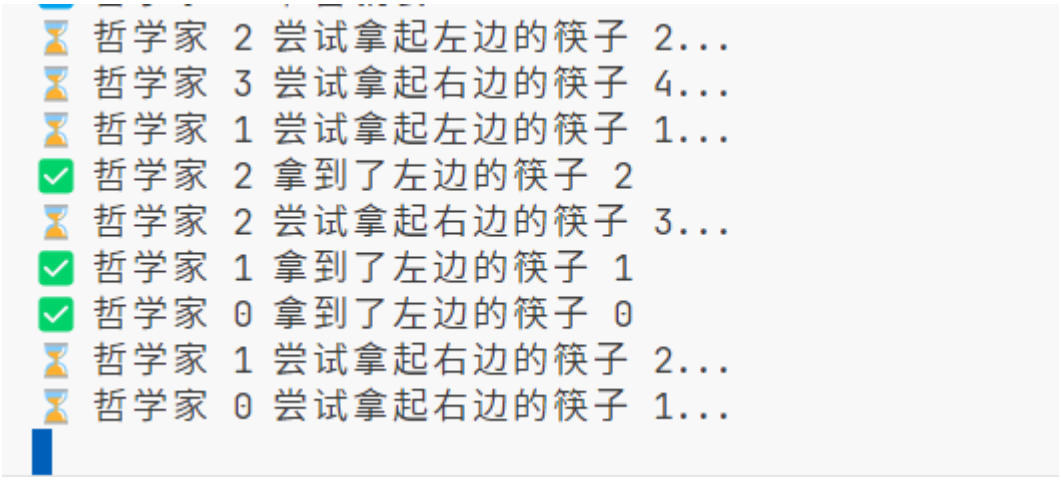
```
    // 等待所有子进程结束
    for (i, &pid) in children.iter().enumerate() {
        if pid != 0 {
            let status = sys_wait_pid(pid);
            crate::println!("哲学家 {} (PID: {}) 退出，状态码 {} 💫", i, pid,
status);
        }
    }

    // 清理信号量
    for i in 0..PHILOSOPHER_COUNT {
        if !CHOPSTICKS[i].remove() {
            crate::println!("清理筷子{}失败 ×", i);
        }
    }

    0
}

entry!(main);
```

2. 尝试构造**饥饿情况**，即某些哲学家无法获得足够的机会就餐。

思路：使用 START_SEM 信号量（初始化为 0）来同步所有哲学家的开始时间：

所有哲学家进程创建后都在等待 START_SEM 主进程创建完所有子进程后，释放 PHILOSOPHER_COUNT 次信号量 这样所有哲学家几乎同时开始执行。

成功实现（这里修改了图标便于区分和观察：

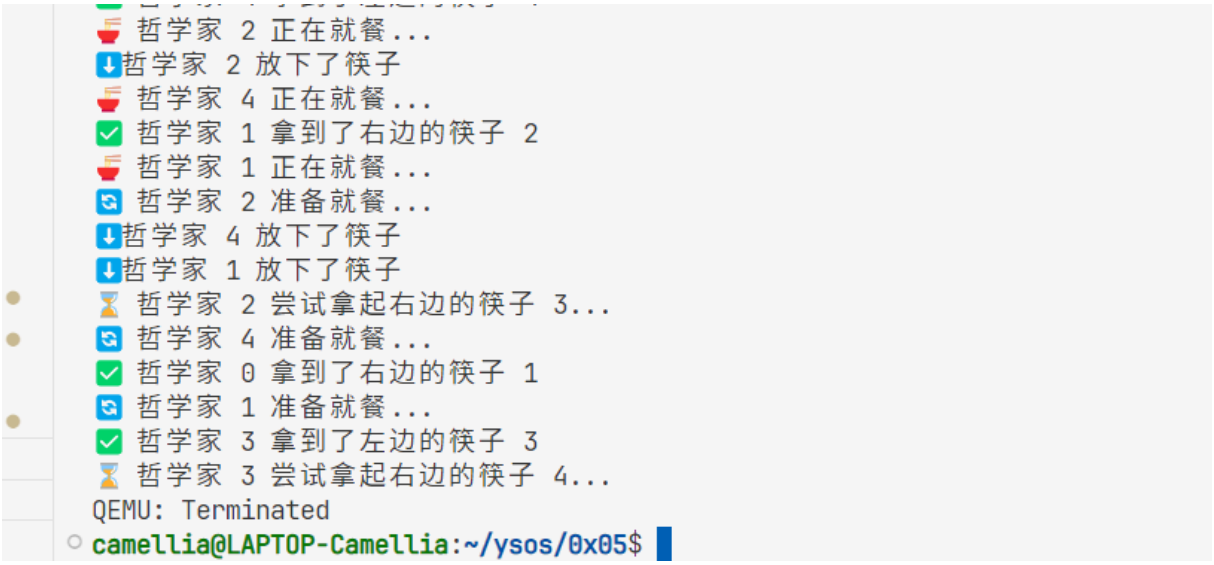


解决方案：

实现奇偶策略，主要修改 philosopher 函数中拿筷子的逻辑：

偶数号哲学家先拿右边的筷子，再拿左边的筷子 奇数号哲学家先拿左边的筷子，再拿右边的筷子 这样可以打破循环等待条件。

成功解决：



3. 尝试构造饥饿情况，即某些哲学家无法获得足够的机会就餐。

我们可以:

重新引入随机数来控制就餐时间 让某些哲学家（比如编号 0 和 2）的就餐时间明显更长 保持奇偶策略避免死锁 增加计数器记录每个哲学家的就餐次数 这样可以观察到某些哲学家因为邻座总是长时间占用筷子而很难获得就餐机会。

结果：从输出可以明显看到饥饿现象:

- 偶数号哲学家的就餐次数明显更多:
  - 哲学家 4：520+次就餐
  - 哲学家 2：450+次就餐
  - 哲学家 0：370+次就餐
  - 奇数号哲学家的就餐次数较少:
- 哲学家 3：440+次就餐
- 哲学家 1：370+次就餐

五. 思考题

1. 1. 输入缓冲区 Mutex 保护问题

1.1. 问题

在 Lab 2 中设计输入缓冲区时，如果不使用无锁队列实现，而选择使用 Mutex 对一个同步队列进行保护，在编写相关函数时需要注意什么问题？考虑在进行 pop 操作过程中遇到串口输入中断的情形，尝试描述遇到问题的场景，并提出解决方案。

1.2. 回答

如果使用 Mutex 保护同步队列，需要注意以下问题：

- **死锁风险**：中断处理程序可能尝试获取已被主线代码（如 `pop` 操作）持有的 Mutex，如果主线代码在持有锁时被中断，且中断处理程序也需要这个锁，就会发生死锁。
- **性能开销**：Mutex 的获取和释放本身有性能开销，在高频率中断或访问的场景下可能成为瓶颈。
- **中断延迟**：如果在持有 Mutex 的临界区内禁用了中断来防止死锁，会增加系统的中断响应延迟。
- **优先级反转**：在支持优先级的系统中，如果低优先级任务持有锁，可能阻塞需要该锁的高优先级任务（甚至中断处理程序）。

遇到问题的场景描述（pop 操作中遇到串口中断）：

1. 用户进程调用 `pop` 函数，该函数首先获取了保护输入缓冲区的 Mutex 锁。
2. 在 `pop` 函数持有锁，正在从队列中取出数据并更新队列状态（例如，移动读指针）的过程中，一个串口输入中断发生。
3. 串口中断处理程序被触发，它需要将新接收到的数据放入输入缓冲区，因此它尝试获取同一个 Mutex 锁。
4. 由于 `pop` 函数仍然持有该 Mutex 锁并且尚未释放，中断处理程序会阻塞等待锁的释放。
    - 如果中断处理程序在等待锁时不允许其他中断（或者自身就是高优先级中断），这可能导致系统对其他中断的响应变慢，甚至丢失中断。
    - 如果中断处理程序忙等待（自旋），会持续消耗 CPU 资源，降低系统效率。
    - 最终可能导致输入数据丢失（如果硬件缓冲区溢出）或系统响应迟钝。

解决方案：

- **在临界区禁用中断**：在 `pop` 和 `push`（中断处理程序中调用）访问共享队列的关键代码段（即持有 Mutex 期间），临时禁用中断。操作完成后立即重新启用中断。这样可以防止中断处理程序在主线代码持有锁时尝试获取锁。

```
// 伪代码示例
fn pop(&self) -> Option<u8> {
    let val;
    // 关中断
    x86_64::instructions::interrupts::without_interrupts(|| {
        let _lock = self.mutex.lock(); // 获取锁
        // ... 从队列中取出数据的操作 ...
        val = self.queue.pop_front();
        // ... 更新队列状态 ...
    }); // 锁在此处释放
    // 开中断（如果之前是开启的）
    val
}

// 中断处理程序中的 push
fn push_in_interrupt(&self, data: u8) {
    // 关中断（通常中断处理开始时已关中断，或需要确保）
    x86_64::instructions::interrupts::without_interrupts(|| {
        let _lock = self.mutex.lock();
        // ... 向队列中添加数据的操作 ...
        self.queue.push_back(data);
        // ... 更新队列状态 ...
    });
    // 开中断（中断处理结束前恢复）
}
```

需要注意的是，禁用中断的时间应尽可能短，以减少对系统响应性的影响。

- **使用 `try_lock`**：中断处理程序可以使用 `try_lock` 尝试获取锁。如果获取失败（意味着锁被其他地方持有），中断处理程序可以选择放弃当前数据、将其放入一个临时的、小的、无锁的缓冲区（如果硬件允许），或者设置一个标志让主线任务稍后处理，而不是直接阻塞。这种方法较为复杂，且可能导致数据丢失。
- **设计中断安全的锁或队列操作**：
  - 例如，主线代码使用锁，但中断处理程序操作一个特殊的、小的、通常是无锁的预留空间，主线代码稍后从这个空间安全地转移数据到主队列。
  - 或者，如果队列操作本身可以被分解为更小的原子步骤，并且中断只执行其中一部分兼容的操作。
- **缩短临界区**：精心设计代码，使得持有 Mutex 的时间尽可能短，减少中断发生时锁被持有的概率和时长。

对于串口输入这样的场景，禁用中断通常是最直接且相对简单的保证数据一致性和避免死锁的方法，但必须严格控制禁用中断的范围。

2. 2. fork 内存复制与页表关系

2.1. 问题

在进行 fork 的复制内存的过程中，系统的当前页表、进程页表、子进程页表、内核页表等之间的关系是怎样的？在进行内存复制时，需要注意哪些问题？

2.2. 回答

页表关系： 在 YSOS 的 `fork` 实现中（根据文档描述，它不复制父进程的内存空间，而是共享，类似于 `vfork` 并结合了部分 `fork` 的特性，如独立的栈）：

1. **内核页表 (Kernel Page Table)**：
   - 内核拥有自己独立的、全局的页表，映射了内核代码、数据、以及所有物理内存的某些部分。
   - 当任何进程（包括父进程或子进程）在用户态陷入内核态时（例如通过系统调用或中断），CPU 会切换到内核栈，并使用内核页表来访问内核空间的数据和代码。
   - 内核页表的高地址部分（内核空间）对于所有进程来说通常是共享且相同的，而低地址部分（用户空间）则会根据当前运行的进程动态变化。
2. **父进程页表 (Parent Process Page Table)**：
   - 父进程拥有自己的页表，该页表定义了其用户虚拟地址空间到物理内存的映射。CR3 寄存器在父进程运行时会指向这个页表的物理地址。
3. **子进程页表 (Child Process Page Table)**：
   - 根据实验描述，YSOS 的 `fork` **不复制父进程的内存空间**，父子进程共享大部分内存（代码段、数据段、堆、BSS 段）。这意味着子进程和父进程**共享同一个页表结构**。
   - 文档中提到 `PageTableContext` 中的 `Cr3RegValue` 被 `Arc` 保护，并且 `PageTableContext::fork()` 方法通过 `clone()` 这个 `Arc` 来实现共享。因此，父子进程的页表实际上指向同一个底层的页表物理内存（PML4 表等）。
   - 虽然页表共享，但子进程会拥有自己**独立的栈空间**。这意味着需要在共享的页表中为子进程的栈分配新的物理页面，并建立新的映射。
4. **系统当前页表**：
   - 指 CPU 的 CR3 寄存器当前指向的页表。
   - 在 `fork` 系统调用执行期间，如果是在父进程的上下文中，当前页表是父进程的页表。
   - 当内核完成子进程的创建并准备调度子进程时，会将子进程的上下文（包括其独立的栈指针和设置为 0 的返回值）加载，并将（共享的）页表地址加载到 CR3（如果因为某些原因需要重新加载的话，但因为共享，这个值和父进程的是一样的）。

内存复制时（特指 YSOS 中为子进程复制/创建栈）需要注意的问题：

1. **栈的独立性与分配**：
   - 必须为子进程分配一块新的物理内存作为其用户栈。

- 需要在共享的页表中为这块新的栈物理内存建立映射，选择一个合适的虚拟地址范围。这个虚拟地址范围不能与父进程的栈或其他已存在的内存区域冲突。文档中建议了一种向下连续分配子进程栈的策略。

2. **栈内容的复制**：
   - 父进程当前的用户栈内容需要被完整地复制到子进程新分配的栈空间中。这确保了子进程从 `fork` 返回时，其栈帧状态与父进程调用 `fork` 时的状态一致（除了返回值）。
   - 文档中提到了使用 `core::ptr::copy_nonoverlapping` 进行高效的内存复制，并强调了源和目标内存区域不能重叠。
3. **栈指针的调整**：
   - 子进程的上下文（通常保存在其中断栈帧或进程控制块中）中的栈指针寄存器（如 `rsp`）必须被更新，以指向其新栈的正确位置（通常是复制完父进程栈内容后的栈顶）。
   - 文档中提到 `ProcessInner::fork` 中需要 “calculate the real stack offset” 和 “update `rsp` in interrupt stack frame”。
4. **返回值的设置**：
   - 父进程从 `fork` 调用返回时，应获得子进程的 PID。
   - 子进程从 `fork` 调用返回时，应获得 0。
   - 这需要在各自的进程上下文中修改 `rax` 寄存器的值。对于子进程，是在其独立的上下文副本中修改。
5. **共享内存的同步（潜在问题，但实验设计规避了部分）**：
   - 由于代码段、数据段、BSS 段和堆（如果 `fork` 前已分配）是共享的，父子进程对这些区域的修改会相互影响。
   - 实验要求 `fork` 在任何堆分配前进行，这极大地简化了堆共享的问题，避免了复杂的堆状态同步和损坏。
6. **页表项权限**：
   - 为子进程新栈分配和映射页面时，要确保页表项具有正确的读写权限。
7. **页表引用计数**：
   - 由于页表结构（`PageTableContext`）通过 `Arc` 共享，当进程退出时，只有最后一个持有该页表引用的进程（父或子）才会真正释放页表所占用的物理帧。这通过 `Arc` 的引用计数自动管理。
8. **地址空间耗尽/碎片化**：
   - 在为子进程分配栈的虚拟地址时，需要一种策略来寻找合适的、足够大的连续虚拟地址空间。如果盲目分配，可能会导致虚拟地址空间碎片化或快速耗尽。文档中提到了当 `map_range` 失败时尝试不同偏移量的策略。

3. 3. fork 与堆分配顺序

3.1. 问题

为什么在实验的实现中，fork 系统调用必须在任何 Rust 内存分配（堆内存分配）之前进行？如果在堆内存分配之后进行 fork，会有什么问题？

3.2. 回答

在实验的 YSOS 实现中，`fork` 系统调用被设计为父子进程共享内存空间（代码段、数据段、BSS 段、堆等），而不复制这些区域，仅为子进程创建独立的栈。这种设计类似于 `vfork` 的行为。

如果在堆内存分配之后进行 `fork`，会出现以下严重问题：

1. **堆状态的破坏 (Heap Corruption)**：
   - Rust 的堆内存分配（如通过 `Box`，`Vec`，`String` 等）依赖于一个全局分配器（如 `alloc` crate 中的默认分配器）。这个分配器内部维护着复杂的元数据来跟踪已分配的内存块、空闲内存块、free lists、bin 等。
   - 如果在 `fork` 之前，父进程已经在堆上分配了内存，那么分配器的内部状态（元数据）以及已分配的堆区域都会被父子进程共享。
   - 当 `fork` 之后，父进程和子进程都可能尝试进行新的堆分配（`alloc`）或释放（`dealloc`）操作。由于它们操作的是同一个共享的分配器状态和堆区域，但彼此并不知道对方的操作，这会导致竞态条件和状态不一致：
     - 两个进程可能同时认为某个空闲块可用，并都尝试分配它。
     - 一个进程释放的内存块可能被另一个进程错误地继续使用（use-after-free），或者被另一个进程再次释放（double-free）。
     - 分配器内部的链表指针或其他数据结构可能被并发修改而损坏，导致分配器无法正常工作。
   - 最终结果是堆的完整性被破坏，程序可能会出现内存泄漏、分配到错误的内存、数据损坏，甚至直接崩溃。
2. **所有权和生命周期问题**：
   - Rust 的核心特性是其所有权系统，它保证了内存安全（例如，没有悬垂指针，没有数据竞争）。一个值在任何时候只有一个所有者，当所有者离开作用域时，值会被 `drop`，其资源会被释放。
   - 如果在 `fork` 前，父进程创建了一个堆上的对象（例如 `Box<T>`），那么在 `fork` 后，父进程和子进程都会认为自己拥有一个指向**同一块堆内存**的 `Box<T>`。
   - 这就破坏了单一所有权原则。如果 `T` 实现了 `Drop` trait：
     - 当父进程中的 `Box<T>` 离开作用域时，它会尝试 `drop` `T` 并释放堆内存。
     - 当子进程中的 `Box<T>` 离开作用域时，它也会尝试 `drop` `T` 并释放**同一块**堆内存。
     - 这会导致 double-free，是严重的内存安全问题。
   - 即使不考虑 `drop`，如果一个进程修改了共享堆对象の内容，另一个进程也会看到这个修改，这可能是非预期的。如果一个进程释放了内存，另一个进程持有的指针就变成了悬垂指针。

3. **标准库和外部库的不可预知行为**:
   - 许多 Rust 标准库和第三方库都依赖于堆分配。如果在 `fork` 前使用了这些库并进行了堆分配，`fork` 后这些库在父子进程中的状态可能会变得不一致或损坏。

为什么实验要求 `fork` 在堆分配前进行? 通过这个限制, 可以确保在 `fork` 时, 堆是“干净”的, 或者说, 父子进程共享的是一个尚未被 (当前进程的生命周期内) 使用的堆。这样，`fork` 之后，如果父子进程各自开始进行堆分配, 它们虽然理论上共享同一个分配器, 但由于实验可能没有实现复杂的并发堆管理，这种共享仍然是危险的。更安全的做法（如果允许 `fork` 后使用堆）是为子进程提供一个全新的、独立的堆，或者实现写时复制（COW）机制来复制堆页面。

实验中的设计选择共享内存（除栈外）是为了简化 `fork` 的实现，避免了大量内存复制的开销，但代价就是对 `fork` 的使用施加了严格的限制，例如必须在堆分配前调用。

4. 4. 原子操作 Ordering 参数

4.1. 问题

进行原子操作时候的 Ordering 参数是什么? 此处 Rust 声明的内容与 C++20 规范 中的一致，尝试搜索并简单了解相关内容，简单介绍该枚举的每个值对应于什么含义。

4.2. 回答

在 Rust (以及 C++20) 中, 原子操作的 `Ordering` 参数(`std::sync::atomic::Ordering`)用于指定该原子操作的内存排序约束。这些约束定义了原子操作如何与程序中其他的内存访问 (包括其他原子操作和非原子操作) 进行排序, 以及它们对其他线程的可见性。这对于在多核处理器上编写正确的并发代码至关重要。

以下是 `Ordering` 枚举各主要值的含义:

1. **`Relaxed` (轻松序)**
   - **含义**: 这是最弱的内存排序。它只保证单个原子操作本身的原子性 (即不会被其他线程的操作打断)，但不对该操作与其他内存读写操作的顺序提供任何额外的保证。编译器和处理器可以自由地将此原子操作与周围的其他内存访问 (包括其他 `Relaxed` 原子操作) 进行重排序。
   - **用途**: 适用于那些只需要保证操作原子性, 而不需要同步其他共享数据或建立特定执行顺序的场景，例如简单的计数器递增（如果计数值本身不用于同步其他状态)。性能开销最小。
2. **`Release` (释放序)**
   - **含义**: 当一个原子操作使用 `Release` 排序时，在此操作之前的所有内存写入 (包括原子和非原子写入) 都不能被重排序到此操作之后。这意味着，当前线程在该 `Release` 操作之前所做的所有内存修改, 对于之后在其他线程中对**同一个原子变量**执行 `Acquire` 操作的线程来说，都是可见的。它起到了“发布”数据的作用。

- **用途**：通常用于生产者线程。当生产者准备好一块共享数据后，它会通过一个 `Release` 存储操作（例如 `store` 或 `fetch_add`）来更新一个标志或指针，告知消费者数据已就绪。

3. `Acquire` **(获取序)**
   - **含义**：当一个原子操作使用 `Acquire` 排序时，在此操作之后的所有内存读取（包括原子和非原子读取）都不能被重排序到此操作之前。这意味着，当前线程在该 `Acquire` 操作之后读取的内存值，能够看到其他线程在对**同一个原子变量**执行 `Release` 操作之前所做的所有内存修改。它起到了“获取”数据的作用。
   - **用途**：通常用于消费者线程。消费者通过一个 `Acquire` 加载操作（例如 `load` 或 `fetch_add`）来检查生产者是否已通过 `Release` 操作发布了数据。如果获取成功，则可以安全地访问那些被发布的数据。
4. `AcqRel` **(Acquire/Release, 获取释放序)**
   - **含义**：此排序同时具有 `Acquire` 和 `Release` 的语义。即，该操作之前的所有内存读写不能被重排序到其后（Release 语义），并且该操作之后的所有内存读写不能被重排序到其前（Acquire 语义）。
   - **用途**：通常用于需要同时读取旧值并写入新值的原子操作，即读-改-写（RMW）操作，例如 `compare_exchange`、`fetch_add`、`swap` 等。这些操作既消费（获取）了原子变量的旧状态，又发布（释放）了它的新状态。
5. `SeqCst` **(Sequentially Consistent, 顺序一致性)**
   - **含义**：这是最强的内存排序，提供了最高的保证。所有标记为 `SeqCst` 的原子操作看起来像是按照一个单一的、全局的总顺序执行的，所有线程都会观察到这个相同的顺序。它不仅包含了 `AcqRel` 的所有保证，还额外禁止了某些在更弱排序模型下可能发生的、与程序顺序不一致的重排序（特别是 `SeqCst` load 不能被重排到 `SeqCst` store 之后，以及 `SeqCst` store 不能被重排到 `SeqCst` load 之前，即使它们操作不同的内存位置）。
   - **用途**：当需要最简单的推理模型，或者当其他较弱的排序不足以保证算法正确性时使用。它是许多原子操作的默认排序，但通常也是性能开销最大的，因为它限制了编译器和处理器的优化空间。

在单核处理器上，由于不存在真正的并行执行和指令重排序问题（相对于其他核心），`Ordering` 的选择通常不会影响程序的最终行为（除了 `Relaxed` 可能允许一些单核内的重排序，但原子性依然保证）。然而，在多核系统中，不正确地使用 `Ordering` 会导致难以调试的并发 bug，如数据竞争、可见性问题和不一致的状态。

5. 5. SpinLock 的 Sync 与 Send Trait

5.1. 问题

在实现 SpinLock 的时候，为什么需要实现 Sync trait? 类似的 Send trait 又是什么含义?

5.2. 回答

`Sync` Trait

- **含义**：`Sync` 是一个标记 trait (marker trait)。如果一个类型 `T` 实现了 `Sync` (即 `T: Sync`)，则意味着对该类型 `T` 的一个**共享引用** `&T` 可以安全地在多个线程之间共享。换句话说，如果 `T` 是 `Sync` 的，那么多个线程可以同时持有对同一个 `T` 实例的共享引用（`&T`）并访问它，而不会导致数据竞争或其他不安全行为。
- **为什么 `SpinLock` 需要实现 `Sync`**：
  - `SpinLock` 的核心目的是允许多个线程通过它来互斥地访问共享数据。这意味着 `SpinLock` 实例本身通常会被多个线程所共享。例如，一个常见的模式是：

    ```
    static MY_LOCK: SpinLock = SpinLock::new();
    // 多个线程会通过 &MY_LOCK 来调用 acquire() 和 release()
    ```

  - 当多个线程调用 `MY_LOCK.acquire()` 时，它们实际上是通过共享引用 `&SpinLock` 来访问同一个 `SpinLock` 实例。
  - `SpinLock` 内部通常使用原子类型（如实验中的 `AtomicBool`）来管理锁的状态。`AtomicBool` 本身是 `Sync` 的，因为它的操作是原子的，可以安全地被多个线程并发访问。
  - 由于 `SpinLock` 通过这些原子操作正确地实现了互斥逻辑，确保了即使多个线程同时访问 `SpinLock` 的方法（如 `acquire`），其内部状态的改变也是安全的，并且能正确地序列化对受保护资源的访问。因此，`SpinLock` 类型本身可以安全地被多线程共享，故它应该实现 `Sync`。
  - 文档中给出的 `unsafe impl Sync for SpinLock {}` 是开发者在确认 `SpinLock` 的内部实现（基于 `AtomicBool` 和正确的原子操作）确实保证了通过共享引用进行并发访问的安全性后，向编译器做出的断言。如果 `SpinLock` 内部包含了非 `Sync` 的数据（例如 `std::cell::Cell`）且没有通过其他同步机制妥善保护，那么这个 `unsafe impl` 就是不正确的。

`Send` Trait

- **含义**：`Send` 也是一个标记 trait。如果一个类型 `T` 实现了 `Send` (即 `T: Send`)，则意味着 `T` 类型的值可以安全地从一个线程**转移所有权**到另一个线程。例如，可以将一个 `T` 类型的值作为参数传递给 `std::thread::spawn` 创建的新线程。
- **`SpinLock` 和 `Send`**：
  - 一个 `SpinLock` 实例通常也是 `Send` 的。因为其内部状态（如 `AtomicBool`）是 `Send` 的，所以 `SpinLock` 结构体本身也可以安全地将其所有权从一个线程转移到另一个线程。
  - 例如，你可以在一个线程中创建一个 `SpinLock`，然后将这个 `SpinLock` 移动到另一个线程中去使用。
  - Rust 的编译器通常会自动推导一个类型是否为 `Send` 或 `Sync`。如果一个结构体的所有字段都是 `Send`，那么该结构体通常也是 `Send` 的。类似地，如果所有字段都是 `Sync`，

结构体通常也是 `Sync` 的（除非使用了像 `*mut T` 这样的裸指针，它们默认不是 `Send` / `Sync`）。

- 如果需要，也可以使用 `unsafe impl Send for SpinLock {}`，前提是开发者确信转移 `SpinLock` 的所有权到另一个线程是安全的。

总结关系：

- `T: Send` 意味着 `T` 类型的值可以被安全地移动到另一个线程。
- `T: Sync` 意味着 `&T` (对 `T` 的共享引用) 可以被安全地用于多个线程之间的共享访问。
- 一个重要的推论是：如果 `T: Sync`，那么 `&T` 必须是 `Send`。因为如果 `&T` 可以在线程间共享，那么你必须能够将这个引用发送给其他线程。反之不一定成立（例如 `MutexGuard` 是 `Send` 但不是 `Sync`）。

对于 `SpinLock` 来说，它既需要被多个线程共享访问（所以需要 `Sync`），通常也可以将其所有权转移到其他线程（所以通常也是 `Send`）。

6. 6. pause 与 hlt 指令对比

6.1. 问题

`core::hint::spin_loop` 使用的 `pause` 指令和 Lab 4 中的 `x86_64::instructions::hlt` 指令有什么区别？这里为什么不能使用 `hlt` 指令？

6.2. 回答

`pause` 指令 (通常由 `core::hint::spin_loop()` 在 x86/x86_64 架构上编译产生)

- **区别**：
  1. **目的与行为**：`pause` 指令是给处理器的一个提示 (hint)，表明当前代码正在执行一个自旋等待循环 (spin-wait loop)，即在一个循环中反复检查某个条件是否满足。它并**不会**使处理器停止执行指令或进入低功耗状态。
  2. **性能优化**：在支持超线程 (Hyper-Threading) 或 SMT 的处理器上，`pause` 指令可以提高性能。当一个逻辑处理器在自旋时，`pause` 可以减少它对执行资源的占用，从而让同一物理核心上的其他逻辑处理器有更多资源可用。它还可以避免因错误的内存顺序推测而导致的流水线冲刷惩罚。
  3. **功耗**：相比于一个没有任何操作的紧密循环，`pause` 可以略微降低功耗，但远不如 `hlt`。
  4. **延迟**：`pause` 会引入一个非常小的、依赖于具体处理器型号的延迟。
  5. **唤醒**：线程执行 `pause` 后仍然是活动的，它会继续执行循环中的下一条指令（通常是再次检查条件）。它不需要外部中断来“唤醒”。
- **使用场景**：专门用于自旋锁的等待循环中，当锁不可用时，线程在循环里调用 `pause` 并不断检查锁的状态。

`hlt` (Halt) 指令 (`x86_64::instructions::hlt()`)

- **区别**:
  1. **目的与行为**: `hlt` 指令使处理器停止执行指令，并进入一个低功耗的暂停状态。CPU 将不再从内存中取指和执行。
  2. **功耗**: 显著降低处理器功耗。
  3. **唤醒**: 处理器会保持在 `hlt` 状态，直到接收到一个外部中断 (如硬件中断 NMI, INTR)、调试异常、或 RESET 信号。当中断被处理完毕后，如果中断返回指令 (`iret`) 指向 `hlt` 之后的指令，CPU 才会继续执行。
  4. **线程状态**: 执行 `hlt` 的线程会变为非活动状态，完全依赖外部事件来恢复执行。
- **使用场景**: 当操作系统没有任务可运行时 (例如, 所有可运行进程都在等待 I/O 或其他事件)，调度器会执行 `hlt` 指令让 CPU 进入空闲状态，以节省能源，直到下一个时钟中断或其他中断发生，触发调度器重新检查是否有任务可运行。

为什么在自旋锁的等待循环中不能使用 `hlt` 指令?

1. **违背自旋锁设计初衷**: 自旋锁的核心思想是, 当锁的持有时间非常短时, 线程通过“自旋”(忙等待)来等待锁的释放, 以避免昂贵的上下文切换开销。如果使用 `hlt` , 线程会立即放弃 CPU 并进入睡眠状态，等待外部中断唤醒。这引入了中断延迟和可能的上下文切换，完全失去了自旋锁“快速检查、快速获取”的优势。获取锁的延迟会大大增加。
2. **死锁风险（尤其单核或中断禁用时）**:
   - 如果一个线程 A 尝试获取锁，发现锁被线程 B 持有，然后线程 A 执行了 `hlt` 。如果系统是单核的，或者在多核但中断被禁用的情况下，线程 B 可能永远没有机会运行并释放锁，导致线程 A 永久 `hlt` ，形成死锁。
   - 即使在多核系统中，线程 A `hlt` 后也无法主动检查锁的状态，它完全依赖于某个中断的发生。这个中断可能与锁的释放毫无关系，或者在锁释放后很久才发生。
3. **不必要的唤醒延迟**: 即使锁很快被释放了，执行 `hlt` 的线程也必须等到下一个中断 (例如时钟中断) 才能被唤醒，然后才能再次检查锁的状态。这引入了不必要的延迟。而使用 `pause` 的自旋循环可以更及时地发现锁已被释放。
4. **语义不符**:
   - 自旋锁是“忙等待”(busy-waiting)：线程保持活动状态，消耗少量 CPU 资源（通过 `pause` 优化)，并积极、反复地检查锁是否可用。
   - `hlt` 则是“阻塞等待”或“睡眠等待”(sleep-waiting)：线程放弃 CPU，不消耗 CPU 资源，等待外部事件通知。

因此， `pause` 是对自旋等待的微小优化，使其在等待期间对系统更友好，但线程依旧保持轮询状态。而 `hlt` 会使线程完全停止，不适用于需要快速响应锁释放的自旋锁场景。

六. 代码附录：修改过的最新版的 `sys::fork` 和 `stack::fork` 以及 `ProcessInner::fork` 代码、`mq` 测试代码

1. sys::fork

```
pub fn sys_fork(context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();

        manager.save_current(context); // Save current process as parent

        let child_pid = manager.fork(); // Create child process

        // Set return value for parent process (child's PID)
        context.set_rax(child_pid.0 as usize);

        manager.push_ready(child_pid); // Push child to ready queue
                                       // Parent is handled by save_current

        let next_pid = manager.switch_next(context); // Switch to next process

        // If we switched to child process, set its return value to 0
        if next_pid == child_pid {
            context.set_rax(0);
        }
    });
}
```

2. stack::fork

```
pub fn fork(
        &self,
        mapper: MapperRef,
        alloc: FrameAllocatorRef,
        stack_offset_count: u64, // Number of existing children, used to offset
the new stack
```

```
    ) -> Self {
        // 1. Calculate new stack range for the child
        // Start with a base offset and try to find a free stack range
        let mut offset = stack_offset_count + 1; // Add 1 to ensure we don't
overlap with parent
        let mut new_stack_range = None;
        let mut attempts = 0;

        while attempts < 10 { // Limit attempts to prevent infinite loop
            let current_addr = STACK_MAX - (offset * STACK_MAX_SIZE);

            if current_addr < STACK_MAX_SIZE { // Arbitrary lower bound to
prevent issues
                panic!("Out of stack space for new process");
            }

            let top_page =
Page::<Size4KiB>::containing_address(VirtAddr::new(current_addr));
            let start_page = top_page - self.usage + 1;
            let try_range = Page::range(start_page, top_page + 1);

            let mut range_is_free = true;
            // Check if any page in range is already mapped
            for page in try_range.clone() {
                if unsafe { mapper.translate_page(page).is_ok() } {
                    range_is_free = false;
                    break;
                }
            }

            if range_is_free {
                new_stack_range = Some(try_range);
                break;
            }

            offset += 1;
            attempts += 1;
        }

        let new_stack_range = new_stack_range.expect("Failed to find free stack
space after 10 attempts");

        // 2. Allocate and map new stack for child
```

```
        // Map the free range we found
        let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE |
PageTableFlags::USER_ACCESSIBLE;
        for page in new_stack_range.clone() {
            let frame = alloc
                .allocate_frame()
                .ok_or(MapToError::<Size4KiB>::FrameAllocationFailed)
                .expect("Stack fork: Frame allocation failed for child stack");
            unsafe {
                mapper
                    .map_to(page, frame, flags, alloc)
                    .expect("Stack fork: Failed to map child stack page")
                    .flush();
            }
        }

        // 3. Copy the *entire stack* from parent to child
        let parent_stack_bottom_addr = self.range.start.start_address().as_u64();
        let child_stack_bottom_addr =
new_stack_range.start.start_address().as_u64();

        Self::clone_range(parent_stack_bottom_addr, child_stack_bottom_addr,
self.usage);

        // 4. Return the new stack
        Self {
            range: new_stack_range,
            usage: self.usage, // Child stack initially has the same usage as
parent
        }
    }
```

3. mq/src/main.rs

```
#![no_std]
#![no_main]

use lib::*;

extern crate lib;

// 消息队列容量，可以设置为1, 4, 8, 16进行测试
const QUEUE_CAPACITY: usize = 16;
```

```rust
// 进程总数
const PROCESS_COUNT: usize = 16;
// 每个进程处理的消息数量
const MESSAGE_COUNT: usize = 10;
// 生产者数量
const PRODUCER_COUNT: usize = PROCESS_COUNT / 2;
// 消费者数量
const CONSUMER_COUNT: usize = PROCESS_COUNT / 2;

// 信号量键值
const MUTEX_SEM_KEY: u32 = 0x1000;        // 互斥锁信号量
const EMPTY_SEM_KEY: u32 = 0x1001;        // 空槽位信号量
const FILLED_SEM_KEY: u32 = 0x1002;       // 满槽位信号量
const PRINT_MUTEX_KEY: u32 = 0x1003;      // 打印互斥锁

// 信号量
static MUTEX_SEM: Semaphore = Semaphore::new(MUTEX_SEM_KEY);
static EMPTY_SEM: Semaphore = Semaphore::new(EMPTY_SEM_KEY);
static FILLED_SEM: Semaphore = Semaphore::new(FILLED_SEM_KEY);
static PRINT_MUTEX: Semaphore = Semaphore::new(PRINT_MUTEX_KEY);

// 消息队列
static mut QUEUE: [u32; QUEUE_CAPACITY] = [0; QUEUE_CAPACITY];
static mut QUEUE_HEAD: usize = 0;
static mut QUEUE_TAIL: usize = 0;
static mut QUEUE_SIZE: usize = 0;

fn main() -> isize {
    println!("消息队列测试开始，队列容量: {}", QUEUE_CAPACITY);

    // 初始化信号量
    MUTEX_SEM.init(1);                    // 互斥锁初始值为1
    EMPTY_SEM.init(QUEUE_CAPACITY);       // 空槽位初始值为队列容量
    FILLED_SEM.init(0);                   // 满槽位初始值为0
    PRINT_MUTEX.init(1);                  // 打印互斥锁初始值为1

    println!("信号量初始化完成");

    let mut pids = [0u16; PROCESS_COUNT];
```

```
    // 创建生产者进程
    for i in 0..PRODUCER_COUNT {
        let pid = sys_fork();
        if pid == 0 {
            // 子进程
            producer(i);
            sys_exit(0);
        } else {
            // 父进程
            pids[i] = pid;
        }
    }
    // 创建消费者进程
    for i in 0..CONSUMER_COUNT {
        let pid = sys_fork();
        if pid == 0 {
            // 子进程
            consumer(i);
            sys_exit(0);
        } else {
            // 父进程
            pids[PRODUCER_COUNT + i] = pid;
        }
    }
    // 输出所有进程信息
    println!("所有进程创建完成，进程ID: {:?}", pids);
    sys_stat();

    // 等待所有子进程退出
    for pid in pids.iter() {
        if *pid != 0 {
            sys_wait_pid(*pid);
        }
    }

    // 输出最终消息队列的消息数量
    unsafe {
        let queue_size = QUEUE_SIZE;
        println!("所有进程已退出，最终消息队列的消息数量: {}", queue_size);
        if queue_size == 0 {
            println!("测试通过! 队列为空，所有消息都被消费了。");
        } else {
            println!("测试失败! 队列不为空，还有{}条消息未被消费。", queue_size);
        }
    }
```

```
    // 清理信号量
    MUTEX_SEM.remove();
    EMPTY_SEM.remove();
    FILLED_SEM.remove();
    PRINT_MUTEX.remove();

    0
}

// 生产者函数
fn producer(id: usize) {
    let pid = sys_get_pid();

    safe_print(&format!("生产者 #{} (PID: {}) 开始生产消息", id, pid));

    for i in 0..MESSAGE_COUNT {
        // 等待空槽位
        EMPTY_SEM.wait();

        // 获取互斥锁
        MUTEX_SEM.wait();

        // 生产消息
        let message = (id as u32) * 100 + i as u32;
        unsafe {
            QUEUE[QUEUE_TAIL] = message;
            QUEUE_TAIL = (QUEUE_TAIL + 1) % QUEUE_CAPACITY;
            QUEUE_SIZE += 1;

            let queue_size = QUEUE_SIZE;
            safe_print(&format!(
                "生产者 #{} (PID: {}) 生产消息: {}，队列大小: {}/{}，队列是否满: {}",
                id, pid, message, queue_size, QUEUE_CAPACITY, queue_size ==
QUEUE_CAPACITY
            ));
        }

        // 释放互斥锁
        MUTEX_SEM.signal();

        // 通知有新消息
        FILLED_SEM.signal();
        // 短暂延时，模拟生产过程
        delay();
    }
```

```
    safe_print(&format!("生产者 #{} (PID: {}) 完成生产，退出", id, pid));
}

// 消费者函数
fn consumer(id: usize) {
    let pid = sys_get_pid();

    safe_print(&format!("消费者 #{} (PID: {}) 开始消费消息", id, pid));

    for _ in 0..MESSAGE_COUNT {
        // 等待有消息
        FILLED_SEM.wait();

        // 获取互斥锁
        MUTEX_SEM.wait();

        // 消费消息
        let message;
        unsafe {
            message = QUEUE[QUEUE_HEAD];
            QUEUE_HEAD = (QUEUE_HEAD + 1) % QUEUE_CAPACITY;
            QUEUE_SIZE -= 1;

            let queue_size = QUEUE_SIZE;
            safe_print(&format!(
                "消费者 #{} (PID: {}) 消费消息: {}，队列大小: {}/{}，队列是否空: {}",
                id, pid, message, queue_size, QUEUE_CAPACITY, queue_size == 0
            ));
        }

        // 释放互斥锁
        MUTEX_SEM.signal();

        // 通知有空槽位
        EMPTY_SEM.signal();

        // 短暂延时，模拟消费过程
        delay();
    }

    safe_print(&format!("消费者 #{} (PID: {}) 完成消费，退出", id, pid));
}

// 安全打印函数，使用互斥锁保证打印不会被打断
fn safe_print(s: &str) {
```

```
    PRINT_MUTEX.wait();
    println!("{}", s);
    PRINT_MUTEX.signal();
}

// 延时函数
#[inline(never)]
fn delay() {
    for _ in 0..0x10000 {
        core::hint::spin_loop();
    }
}

entry!(main);
```