



# 课程实验报告

## 0x05 fork 的实现、并发与锁机制

课程名称	操作系统原理实验
专业名称	计算机科学与技术
学生姓名	陈政宇
学生学号	23336003
实验地点	东校园-实验中心大楼 B201
实验成绩	
实验日期	2025 年 5 月 20 日

# 目录

<b>1 代码实现 .....</b>	<b>3</b>
1.1 fork 的实现 .....	3
1.2 进程的阻塞与唤醒 .....	6
1.2.1 阻塞 .....	6
1.2.2 唤醒 .....	6
1.3 自旋锁的实现 .....	7
1.4 信号量的实现 .....	8
<b>2 代码测试 .....</b>	<b>11</b>
2.1 fork 测试 .....	11
2.2 多线程计数器 .....	11
2.3 消息队列（生产者-消费者问题） .....	13
2.4 哲学家就餐问题 .....	14
<b>3 思考题 .....</b>	<b>18</b>
3.1 q1 .....	18
3.2 q2 .....	19
3.3 q3 .....	20
3.4 q4 .....	20
3.5 q5 .....	21
3.6 q6 .....	22
<b>4 加分项 .....</b>	<b>22</b>
4.1 fish .....	22
4.2 哲学家就餐问题其二 .....	23

# 1 代码实现

## 1.1 fork 的实现

### Task 1

- 系统调用静态函数，并将其委托给 `ProcessManager::fork`。
- `ProcessManager::fork` 负责创建一个新的进程，并将其添加到就绪队列中，并委托 `Process::fork`。

```

1  pub fn vfork(context: &mut ProcessContext) { Rust
2      x86_64::instructions::interrupts::without_interrupts(|| {
3          let manager = get_process_manager();
4          let pid = manager.save_current(context);
5          manager.vfork();
6          manager.push_ready(pid);
7          manager.switch_next(context);
8      })
9  }
10 // ProcessManager::vfork
11 pub fn vfork(&self) {
12     let child = self.current().vfork();
13     let pid = child.pid();
14     self.add_proc(pid, child);
15     self.push_ready(pid);
16     debug!("Current queue: {:?}", self.ready_queue.lock());
17 }
```

### Task 2

`Process::fork` 将具体实现委托给 `ProcessInner::fork`。

```

1  pub fn vfork(self: &Arc<Self>) -> Arc<Self> { Rust
2      let mut inner = self.inner.write();
3      let new_inner = inner.vfork(Arc::downgrade(self));
4      let pid = ProcessId::new();
5      let child = Arc::new(Process {
6          pid,
7          inner: Arc::new(RwLock::new(new_inner)),
```

```

8     });
9     debug!(
10         "Process {}#{}} vforked to {}#{}}",
11         inner.name(),
12         self.pid,
13         child.inner.read().name(),
14         child.pid
15     );
16     inner.context.set_rax(child.pid.0 as usize);
17     inner.children.push(child.clone());
18     inner.pause();
19     child
20 }

```

### Task 3

ProcessInner::fork 负责创建一个新的进程，并为其分配栈空间。

```

1     pub fn vfork(&mut self, parent: Weak<Process>) ->  ProcessInner{
2         let proc_vm = self.vm().vfork(self.children.len() as u64 +
3         1);
4         let offset = proc_vm.stack.stack_offset(&self.vm().stack);
5         let mut child_context = self.context;
6         child_context.set_stack_offset(offset);
7         child_context.set_rax(0);
8         Self {
9             name: self.name.clone(),
10            parent: Some(parent),
11            status: ProgramStatus::Ready,
12            context: child_context,
13            ticks_passed: 0,
14            exit_code: None,
15            children: Vec::new(),
16            proc_vm: Some(proc_vm),
17            proc_data: self.proc_data.clone(),
18        }
19    }
20    //ProcessVm::vfork
21    pub fn vfork(&self, stack_offset: u64) -> Self {

```

```
21     let page_table = self.page_table.fork();
22     let mapper = &mut page_table.mapper();
23     let alloc = &mut *get_frame_alloc_for_sure();
24     Self {
25         page_table: page_table,
26         stack: self.stack.vfork(mapper, alloc, stack_offset),
27         code: Vec::new(),
28         code_usage: 0,
29     }
30 }
31 //Stack::vfork
32 pub fn vfork(&self, mapper: MapperRef, alloc: FrameAllocatorRef,
33 stack_offset: u64) -> Self {
34     let mut new_stack_base =
35     self.range.start.start_address().as_u64() - stack_offset *
36     STACK_MAX_SIZE;
37     while elf::map_pages(
38         new_stack_base,
39         self.usage,
40         mapper,
41         alloc,
42         true,
43     )
44     .is_err() {
45         new_stack_base -= STACK_MAX_SIZE;
46     }
47     debug!("Map new stack: {:#x}", new_stack_base);
48     unsafe {
49         copy_nonoverlapping(
50             self.range.start.start_address().as_u64() as *mut
51             u64,
52             new_stack_base as *mut u64,
53             (self.usage * Size4KiB::SIZE / 8) as usize,
54         );
55         debug!(
56             "Copy stack range {:#x} to {:#x}",
57             self.range.start.start_address().as_u64(),
58             new_stack_base
59         );
60     }
```

```
57         let new_start =  
            Page::containing_address(VirtAddr::new(new_stack_base));  
58         let new_end = new_start + self.usage;  
59         Self {  
60             range: Page::range(new_start, new_end),  
61             usage: self.usage,  
62         }  
63     }
```

## 1.2 进程的阻塞与唤醒

### 1.2.1 阻塞

#### Task 4

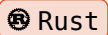
wait\_pid 的实现。

```
1  pub fn wait_pid(pid: ProcessId, context: &mut  
    ProcessContext) { Rust  
2      x86_64::instructions::interrupts::without_interrupts(|| {  
3          let manager = get_process_manager();  
4          if let Some(ret) = manager.get_exit_code(pid) {  
5              context.set_rax(ret as usize);  
6          } else {  
7              manager.wait_pid(pid);  
8              manager.save_current(context);  
9              manager.current().write().block();  
10             manager.switch_next(context);  
11         }  
12     })  
13 }  
14 // ProcessManager::wait_pid  
15 pub fn wait_pid(&self, pid: ProcessId) {  
16     let mut wait_queue = self.wait_queue.lock();  
17     let entry = wait_queue.entry(pid).or_default();  
18     entry.insert(processor::get_pid());  
19 }
```

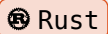
### 1.2.2 唤醒

### Task 5

wake\_up 的实现。

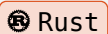
```
1 pub fn wake_up(&self, pid: ProcessId, ret: Option<isize>) {  Rust
2     if let Some(proc) = self.get_proc(&pid) {
3         let mut inner = proc.write();
4         if let Some(ret) = ret {
5             inner.set_return(ret as usize);
6         }
7         inner.pause();
8         self.push_ready(pid);
9     }
10 }
```

进程退出时，唤醒等待它的进程。

```
1 pub fn kill(&self, pid: ProcessId, ret: isize) {  Rust
2     match self.get_proc(&pid) {
3         Some(proc) => {
4             trace!("Kill {:#?}", &proc);
5             proc.kill(ret);
6             if let Some(waiters) =
7                 self.wait_queue.lock().remove(&pid) {
8                 for waiter in waiters {
9                     self.wake_up(waiter, Some(ret));
10                }
11            }
12            None => {
13                warn!("Process #{pid} not found.", pid);
14            }
15        }
16    }
```

## 1.3 自旋锁的实现

### Task 6

```
1 pub struct SpinLock {  Rust
2     bolt: AtomicBool,
```

```
3 }
4
5 impl SpinLock {
6     pub const fn new() -> Self {
7         Self {
8             bolt: AtomicBool::new(false),
9         }
10    }
11
12    pub fn acquire(&self) {
13        // DONE: acquire the lock, spin if the lock is not available
14        while self.bolt.compare_exchange(false, true,
15            Ordering::Acquire, Ordering::Relaxed).is_err() {
16            // spin
17            spin_loop();
18        }
19
20    pub fn release(&self) {
21        // DONE: release the lock
22        self.bolt.store(false, Ordering::Relaxed);
23    }
24 }
```

## 1.4 信号量的实现

### Task 7

```
1  #[derive(Clone, Copy, Debug, PartialEq, Eq, PartialOrd,
2      Ord)]
3  pub struct Semaphore {
4      /* DONE: record the sem key */
5      key: u32,
6  }
7
8  impl Semaphore {
9      pub const fn new(key: u32) -> Self {
10         Semaphore { key }
11     }
12 }
```

 Rust



```
11
12     #[inline(always)]
13     pub fn init(&self, value: usize) -> bool {
14         sys_new_sem(self.key, value)
15     }
16
17     /* DONE: other functions with syscall... */
18     #[inline(always)]
19     pub fn wait(&self) -> bool {
20         sys_sem_wait(self.key)
21     }
22     #[inline(always)]
23     pub fn signal(&self) -> bool {
24         sys_sem_signal(self.key)
25     }
26     #[inline(always)]
27     pub fn free(&self) -> bool {
28         sys_sem_free(self.key)
29     }
30 }
31 //syscalls
32 pub fn sys_sem(args: &SyscallArgs, context: &mut ProcessContext) {
33     match args.arg0 {
34         0 => context.set_rax(sem_new(args.arg1 as u32, args.arg2) as
            usize),
35         1 => sem_wait(args.arg1 as u32, context),
36         2 => sem_signal(args.arg1 as u32, context),
37         3 => context.set_rax(remove_sem(args.arg1 as u32)),
38         _ => context.set_rax(usize::MAX),
39     }
40 }
41 pub fn sem_new(key: u32, value: usize) -> usize {
42     x86_64::instructions::interrupts::without_interrupts(|| {
43         if get_process_manager().current().write().sem_new(key,
            value) {
44             return 0;
45         }
46         1
47     })
48 }
```


```
49
50 pub fn remove_sem(key: u32) -> usize {
51     x86_64::instructions::interrupts::without_interrupts(|| {
52         if get_process_manager().current().write().sem_remove(key) {
53             return 0;
54         }
55         1
56     })
57 }
58
59 pub fn sem_wait(key: u32, context: &mut ProcessContext) {
60     x86_64::instructions::interrupts::without_interrupts(|| {
61         let manager = get_process_manager();
62         let pid = manager.current().pid();
63         let result = manager.current().write().sem_wait(key, pid);
64         match result {
65             SemaphoreResult::Ok => context.set_rax(0),
66             SemaphoreResult::Block(pid) => {
67                 manager.save_current(context);
68                 manager.block(pid);
69                 manager.switch_next(context);
70             },
71             SemaphoreResult::NotExist => context.set_rax(1),
72             _ => unreachable!(),
73         }
74     })
75 }
76
77 pub fn sem_signal(key: u32, context: &mut ProcessContext) {
78     x86_64::instructions::interrupts::without_interrupts(|| {
79         let manager = get_process_manager();
80         let result = manager.current().write().sem_signal(key);
81         match result {
82             SemaphoreResult::Ok => context.set_rax(0),
83             SemaphoreResult::WakeUp(pid) => manager.wake_up(pid,
84                 None),
85             SemaphoreResult::NotExist => context.set_rax(1),
86             _ => unreachable!(),
87         }
88     })
89 }
```

```
88 }
```

## 2 代码测试

### 2.1 fork 测试

写一个简单的 fork 测试程序，创建一个子进程并打印父子进程的 PID。测试结果如下：



```
python ysos.py run

Type 'help' to see available commands.

ysos@machine~
> exec fork
I am the child process, pid: 0
child read value of M: 0xdeadbeef
child changed the value of M: 0x2333
I am the parent process
  PID | PPID | Process Name | Ticks | Status
#  1  | #   0 | kernel      |   221 | Ready
#  2  | #   1 | sh          |   220 | Blocked
#  3  | #   2 | fork        |     1 | Running
Queue : [1]
CPUs  : [0: 3]
Waiting for child to exit...
Child exited with status 64
parent read value of M: 0x2333
process exited with code 1056
ysos@machine~
>
```

### 2.2 多线程计数器

#### Task 8

在所给代码的 pkg/app/counter 中实现了一个多线程计数器，多个线程对一个共享的计数器进行累加操作，最终输出计数器的值。

为了提供足够大的可能性来触发竞态条件，该程序使用了一些手段来刻意构造一个临界区，这部分代码不应被修改。

你需要通过上述两种方式，分别保护该临界区，使得计数器的值最终为 800。

通过使用信号量保护临界区。关键代码如下：

```
1 fn do_counter_inc() {
2     for _ in 0..100 {
3         // FIXME: protect the critical section
4         MUTEX.wait();
5         inc_counter();
6         MUTEX.signal();
7     }
8 }
```

通过使用自旋锁保护临界区。关键代码如下：

```
1 fn do_counter_inc() {
2     for _ in 0..100 {
3         spin_lock.acquire();
4         inc_counter();
5         spin_lock.release();
6     }
7 }
```

测试结果

```
python ysos.py run
# 1 | # 0 | kernel | 41259 | Ready
# 2 | # 1 | sh | 41249 | Blocked
# 5 | # 2 | counter | 8 | Running
# 6 | # 5 | counter | 4 | Ready
# 7 | # 5 | counter | 3 | Blocked
# 8 | # 5 | counter | 2 | Blocked
# 9 | # 5 | counter | 2 | Blocked
# 10 | # 5 | counter | 1 | Blocked
# 11 | # 5 | counter | 1 | Blocked
# 12 | # 5 | counter | 1 | Blocked
# 13 | # 5 | counter | 1 | Blocked
Queue : [1, 6]
CPUs : [0: 5]
#5 waiting for #6...
#5 waiting for #7...
#5 waiting for #8...
#5 waiting for #9...
#5 waiting for #10...
#5 waiting for #11...
#5 waiting for #12...
#5 waiting for #13...
COUNTER result: 800
process exited with code 0
ysos@machine~
```

## 2.3 消息队列（生产者-消费者问题）

### Task 9

假设一个现实中咖啡店的场景：咖啡师以固定的速度制作咖啡，杯子的数量有限，制作好的咖啡同时存在的数量有限。每个顾客都要消费若干咖啡，同一时间能够消费的咖啡数量也有限（受到咖啡师和杯子数量的制约）。

算法流程如下：

- 共享缓冲区 模拟的是杯子架，容量为 `CUP_CAPACITY`。共享变量 `COFFEE_CUPS` 表示当前生产但尚未被消费的咖啡杯数。
- 信号量使用 代码中定义了三个全局信号量，用于协调生产和消费过程：
  - **FULL\_CAPACITY**：初值为杯子架的容量（`CUP_CAPACITY`），用于表明还有空位可以放杯子。生产者（咖啡师）在生产前需等待此信号量，保证杯子架没有满，生产后信号量减 1。
  - **NON\_EMPTY**：初值为 0，用于表示杯子架中至少有一杯咖啡可供消费者取用。消费者在消费前需等待此信号量，消费后释放（加 1）`FULL_CAPACITY`。
  - **MUTEX**：初值为 1，用于在操作共享变量 `COFFEE_CUPS` 时提供互斥保护，保证同一时刻只有一个进程能修改该变量。
- 生产者（咖啡师）流程 在 `coffee_maker()` 函数中：
  1. 循环生产 `TOTAL_PRODUCTION` 杯（等于顾客数量乘以每位顾客的消费杯数）。
  2. 每次生产前先 `wait FULL_CAPACITY`，确保有空位。
  3. 进入临界区（通过 `MUTEX.wait()`）后，将 `COFFEE_CUPS` 增加 1，表示生产了一杯咖啡。
  4. 离开临界区（通过 `MUTEX.signal()`）后，再 `signal NON_EMPTY`，通知消费者有可消费的咖啡。
  5. 在整个生产过程模拟一定的延迟（`delay()`），以便观察生产与消费的交替效果。
- 消费者（顾客）流程 每个在 `customer()` 中的顾客进程：
  1. 循环消费固定杯数（`CONSUME_PER_CUSTOMER`）。
  2. 每次消费前先 `wait NON_EMPTY`，确保杯子架中至少有一杯咖啡可取。
  3. 进入临界区（通过 `MUTEX.wait()`）后，将 `COFFEE_CUPS` 减少 1，表示消费一杯咖啡。
  4. 离开临界区（通过 `MUTEX.signal()`）后，再 `signal FULL_CAPACITY`，通知生产者有空位可生产。

5. 同样，消费过程中也会有延迟，用于模拟实际情况。

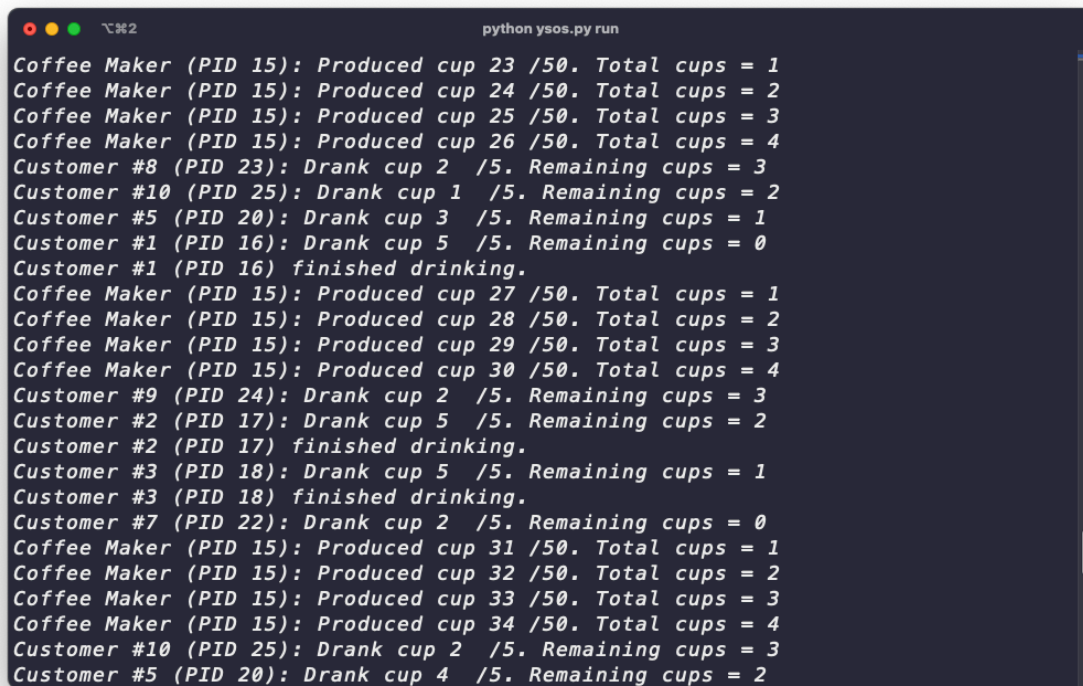
- 进程调度与同步

- 父进程调用 `fork()` 创建 1 个咖啡师和 `TOTAL_CUSTOMERS` 个顾客进程。
- 父进程随后等待所有子进程通过 `sys_wait_pid()` 退出，确保生产与消费过程完整执行后再做后续资源清理。

- 总体算法关键点

1. 边界控制：生产者通过 `FULL_CAPACITY` 控制生产速度，确保不会超出杯子架最大容量；消费者通过 `NON_EMPTY` 保证在有产品时进行消费。
2. 临界区保护：对共享变量 `COFFEE_CUPS` 的修改通过 `MUTEX` 信号量保护，避免多进程并发修改产生竞态条件。
3. 进程间通信：信号量的 `wait()` 与 `signal()` 用于进程之间的同步和协调，使得生产与消费在各自条件满足时正确进行。

测试结果如下：



```
python ysos.py run
Coffee Maker (PID 15): Produced cup 23 /50. Total cups = 1
Coffee Maker (PID 15): Produced cup 24 /50. Total cups = 2
Coffee Maker (PID 15): Produced cup 25 /50. Total cups = 3
Coffee Maker (PID 15): Produced cup 26 /50. Total cups = 4
Customer #8 (PID 23): Drank cup 2 /5. Remaining cups = 3
Customer #10 (PID 25): Drank cup 1 /5. Remaining cups = 2
Customer #5 (PID 20): Drank cup 3 /5. Remaining cups = 1
Customer #1 (PID 16): Drank cup 5 /5. Remaining cups = 0
Customer #1 (PID 16) finished drinking.
Coffee Maker (PID 15): Produced cup 27 /50. Total cups = 1
Coffee Maker (PID 15): Produced cup 28 /50. Total cups = 2
Coffee Maker (PID 15): Produced cup 29 /50. Total cups = 3
Coffee Maker (PID 15): Produced cup 30 /50. Total cups = 4
Customer #9 (PID 24): Drank cup 2 /5. Remaining cups = 3
Customer #2 (PID 17): Drank cup 5 /5. Remaining cups = 2
Customer #2 (PID 17) finished drinking.
Customer #3 (PID 18): Drank cup 5 /5. Remaining cups = 1
Customer #3 (PID 18) finished drinking.
Customer #7 (PID 22): Drank cup 2 /5. Remaining cups = 0
Coffee Maker (PID 15): Produced cup 31 /50. Total cups = 1
Coffee Maker (PID 15): Produced cup 32 /50. Total cups = 2
Coffee Maker (PID 15): Produced cup 33 /50. Total cups = 3
Coffee Maker (PID 15): Produced cup 34 /50. Total cups = 4
Customer #10 (PID 25): Drank cup 2 /5. Remaining cups = 3
Customer #5 (PID 20): Drank cup 4 /5. Remaining cups = 2
```

## 2.4 哲学家就餐问题

### Task 10

假设有 5 个哲学家，他们的生活只是思考和吃饭。这些哲学家共用一个圆桌，每位都有一把椅子。在桌子中央有一碗米饭，在桌子上放着 5 根筷子。

当一位哲学家思考时，他与其他同事不交流。时而，他会感到饥饿，并试图拿起与他相近的两根筷子（筷子在他和他的左或右邻居之间）。

一个哲学家一次只能拿起一根筷子。显然，他不能从其他哲学家手里拿走筷子。当一个饥饿的哲学家同时拥有两根筷子时，他就能吃。在吃完后，他会放下两根筷子，并开始思考。

算法流程如下：

主要数据结构和信号量

- **CHOPSTICKS** 数组 用 `semaphore_array!` 宏声明了 5 根筷子，每根信号量初始值为 1，表示筷子处于空闲状态。每个哲学家用自己的编号决定使用哪两根筷子：左边为 `CHOPSTICKS[id]`，右边为 `CHOPSTICKS[(id+1)%5]`。
- **WAITER** 信号量 仅在正常（以及饥饿）模式下使用，用来限制最多只有 4 个哲学家同时去拿筷子，以避免所有 5 人同时拿起一根筷子而造成死锁（即形成一个循环等待）。
- 延迟函数 使用 `delay` 函数和 `spin_loop()` 模拟不同阶段（思考、就餐）的等待时间，使得各哲学家之间的动作不同步，这样更容易观察到竞争、饥饿或死锁现象。

主函数 (main):

- 输入读取与模式选择 主函数首先读取一行输入，将其解析为一个 u64 数值（demo 模式标识）。
  - 若 `demo = 1`，就表示走死锁示例路径，不使用 **WAITER**。
  - 若 `demo != 1`，则在正常模式或饥饿模式下使用 **WAITER**，初始化值为 4（即 `PHILOSOPHER_COUNT - 1`）。
- 信号量初始化 循环为每根筷子调用 `init(1)`；对模式不同情况下 **WAITER** 信号量也进行初始化。
- 创建哲学家进程 主进程依次调用 `sys_fork()` 创建 5 个子进程，每个子进程传入自己的编号 `id`，并调用 `philosopher(id, demo)` 开始各自的逻辑。
- 等待子进程退出 主进程使用 `sys_wait_pid` 等待所有哲学家进程退出，然后打印完成信息。

哲学家行为函数 **philosopher**:

在每个哲学家进程中，重复进行多次进餐（共 `EAT_TIMES` 次）：



- 思考阶段 根据不同 demo 模式，设置不同的延迟：
  - 在死锁模式 (`demo == 1`)，延迟较短，使得各哲学家近乎同时饥饿；
  - 在正常或饥饿模式下，延迟随机（或基于 id 设定不同延迟，模拟饥饿现象）。
- 就餐阶段 —— 获取筷子：
  - 死锁模式 (**demo == 1**)：所有哲学家直接按照相同顺序操作，即先 wait 自己编号对应的左筷子，再延迟后 wait 右边的筷子。这种统一顺序容易造成环状等待，从而形成死锁。
  - 正常/饥饿模式：利用奇偶不同的顺序进行：
    - 若 id 为偶数，则先拿左筷子、再拿右筷子；
    - 若 id 为奇数，则先拿右筷子、再拿左筷子。

除此之外，在非死锁模式下，还可以用 WAITER 信号量参与协调（代码中在就餐结束后调用 `WAITER.signal()`），确保同时试图拿筷子的哲学家数量不会超过 4，从而打破可能的循环等待。
- 进餐与延迟 成功获得两根筷子后，哲学家打印消息，并调用一次随机延迟模拟进餐时间。
- 释放筷子 就餐结束后，哲学家先后调用 `signal()` 释放自己拥有的两个筷子，并打印放下筷子的日志。在非死锁模式下还会调用 `WAITER.signal()` 释放服务生信号量，允许其他哲学家进入就餐阶段。
- 退出过程 当完成所有就餐次数后，该哲学家打印满意离开的信息，然后调用 `sys_exit(0)` 退出进程。

测试结果如下：

- 正常状态（采用一个服务生，保证不会同时有 5 个哲学家拿起筷子）：



```
python ysos.py run
Philosopher 0 (PID 27) picked up left chopstick 0.
Philosopher 3 (PID 30) picked up right chopstick 4.
Philosopher 3 (PID 30) picked up left chopstick 3.
Philosopher 3 (PID 30) is eating (turn 1).
Philosopher 1 (PID 28) put down chopstick 1.
Philosopher 1 (PID 28) put down chopstick 2.
Philosopher 4 (PID 31) picked up left chopstick 4.
Philosopher 0 (PID 27) picked up right chopstick 1.
Philosopher 0 (PID 27) is eating (turn 1).
Philosopher 2 (PID 29) picked up left chopstick 2.
Philosopher 3 (PID 30) put down chopstick 3.
Philosopher 3 (PID 30) put down chopstick 4.
Philosopher 0 (PID 27) put down chopstick 0.
Philosopher 0 (PID 27) put down chopstick 1.
Philosopher 2 (PID 29) picked up right chopstick 3.
Philosopher 2 (PID 29) is eating (turn 1).
Philosopher 4 (PID 31) picked up right chopstick 0.
Philosopher 4 (PID 31) is eating (turn 1).
Philosopher 2 (PID 29) put down chopstick 2.
Philosopher 2 (PID 29) put down chopstick 3.
Philosopher 4 (PID 31) put down chopstick 4.
Philosopher 4 (PID 31) put down chopstick 0.
Philosopher 1 (PID 28) picked up right chopstick 2.
Philosopher 1 (PID 28) picked up left chopstick 1.
Philosopher 1 (PID 28) is eating (turn 2).
```

- 饥饿的哲学家（第二个哲学家 pid#4 由于动作慢了一些，抢不到筷子，陷入了 Blocked）:

```
python ysos.py run
Philosopher 0 (PID 4) started.
Philosopher 0 (PID 4) picked up chopstick 0.
Philosopher 0 (PID 4) picked up chopstick 1.
Philosopher 0 (PID 4) is eating (turn 1).
Philosopher 1 (PID 5) started.
Philosopher 2 (PID 6) started.
Philosopher 2 (PID 6) picked up chopstick 2.
Philosopher 2 (PID 6) picked up chopstick 3.
Philosopher 2 (PID 6) is eating (turn 1).
Philosopher 3 (PID 7) started.
Philosopher 0 (PID 4) put down chopstick 0.
Philosopher 0 (PID 4) put down chopstick 1.
Philosopher 0 (PID 4) picked up chopstick 0.
Philosopher 4 (PID 8) started.
#3: Created philosophers: [4, 5, 6, 7, 8]
  PID | PPID | Process Name | Ticks | Status
# 1 | # 0 | kernel | 294 | Ready
# 2 | # 1 | sh | 234 | Blocked
# 3 | # 2 | dinner | 59 | Running
# 4 | # 3 | dinner | 5 | Blocked
# 5 | # 3 | dinner | 2 | Ready
# 6 | # 3 | dinner | 3 | Ready
# 7 | # 3 | dinner | 1 | Blocked
# 8 | # 3 | dinner | 1 | Blocked
Queue : [1, 5, 6]
```

- 死锁状态（所有哲学家都在等待对方放下筷子）:

```
python ysos.py run

Philosopher 0 (PID 10) started.
Philosopher 0 (PID 10) picked up left chopstick 0.
Philosopher 1 (PID 11) started.
Philosopher 1 (PID 11) picked up left chopstick 1.
Philosopher 2 (PID 12) started.
Philosopher 2 (PID 12) picked up left chopstick 2.
Philosopher 3 (PID 13) started.
Philosopher 3 (PID 13) picked up left chopstick 3.
Philosopher 4 (PID 14) started.
Philosopher 4 (PID 14) picked up left chopstick 4.
#9: Created philosophers: [10, 11, 12, 13, 14]
  PID | PPID | Process Name | Ticks | Status
# 1 | # 0 | kernel | 23349 | Ready
# 2 | # 1 | sh | 23165 | Blocked
# 9 | # 2 | dinner | 83 | Running
# 10 | # 9 | dinner | 3 | Blocked
# 11 | # 9 | dinner | 3 | Blocked
# 12 | # 9 | dinner | 3 | Blocked
# 13 | # 9 | dinner | 2 | Ready
# 14 | # 9 | dinner | 1 | Ready
Queue : [1, 13, 14]
CPUs : [0: 9]
#9 waiting for philosopher #10...
```

### 3 思考题

#### 3.1 q1

##### ? Question

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. 中断上下文的重入问题 如果在执行 pop 操作期间发生串口中断，而中断处理程序也试图访问同一个队列（例如将新输入的数据 push 到队列中），如果中断处理程序也需要获得相同的 Mutex，就可能导致死锁或陷入无限等待。因为主程序持有锁，而中断处理程序又阻塞在等待锁的获取上，但中断处理程序无法完成执行，从而影响整个系统。
2. 临界区范围与中断禁用 为避免上述问题，通常需要在进入临界区进行队列操作之前禁用中断，这样可以确保在执行 pop 操作期间不会被串口中断打断。完成后，再恢

复中断状态。但这需要特别小心，确保在禁用中断的期间临界区操作足够快，防止长时间禁用中断影响系统响应。

### 3. 设计方案与解决思路

- 禁用中断保护临界区：在执行 pop（或者其他涉及 Mutex 的队列操作）时，临时关闭串口中断及其他可能访问该队列的中断。这样可以防止中断处理程序在已经持有锁后再次尝试获取，从而避免死锁。
- 分离中断处理与主逻辑：另一种思路是让中断处理程序仅完成最小的、快速的工作，比如将数据写入一个预先分配的环形缓冲区，而不直接操作被 Mutex 保护的主队列。主程序定期从该环形缓冲区中提取数据，再将其推入主队列。这样就不会出现中断在持有锁期间尝试访问队列的情况。
- 采用无锁队列或者原子操作：虽然题目要求使用 Mutex，但可以考虑基于锁的队列设计中，如果操作比较简单，可以采用无锁算法来减少或避免临界区，这样也能避免中断和 Mutex 之间的潜在冲突。

### 4. 实际场景示例 假设在 pop 操作中，代码如下：

```
1 fn pop_from_queue() -> Option<Data> {  
2     // 禁用中断，防止中断处理程序同样访问队列  
3     disable_interrupts();  
4     let data = QUEUE.lock().pop();  
5     enable_interrupts();  
6     data  
7 }
```

 Rust

这样，当串口中断发生时，由于中断已经被禁用，中断处理程序不会试图获取锁，从而避免进入死锁状态。如果不能使用全局禁用中断，则必须设计一个中断安全的队列（或者采用分离策略，如上所述）。

## 3.2 q2

### ? Question

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？

#### 1. 父进程与子进程的用户页表

- 在 fork 时，父进程的用户态页表会被复制一份给子进程，在本实验中共享相同的物理内存页面。
- 这样节省了复制内存的开销。

## 2. 内核页表与共享区域

- 内核空间通常在每个进程的页表中都有一份映射，但这一部分并不会被复制，而是所有进程共享同一份内核映射。内核部分通常标记为仅内核可访问，保证用户进程不能随意修改。
- 当子进程执行系统调用或进入内核态时，使用的依然是自己页表中相同的内核映射部分，与父进程保持一致。

## 3. 复制过程中需要注意的问题

- 父子进程的隔离性：确保用户页表中共享的部分在写时能正确分离，同时保持内核映射不被误修改。
- 内核结构与资源计数：有些页或者内存区域可能包含内核数据结构或计数器，这些在 fork 后需要保证子进程正确共享或重新初始化（例如文件描述符表、信号处理结构等），而不仅仅是内存页的复制。

## 3.3 q3

### ? Question

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？

## 1. 内存管理器的状态不一致

- Rust 的堆分配器（如 jemalloc 或系统默认的 malloc）通常维护全局状态（如空闲内存块链表、分配统计等）。fork 会复制父进程的整个地址空间，但不会重置子进程中的内存管理器状态。
- 如果父进程在 fork 前已经分配了堆内存，子进程会继承相同的堆状态（例如相同的指针、锁、元数据）。当子进程尝试分配/释放内存时，可能破坏父进程的堆结构，或触发未定义行为（如双重释放）。

## 2. 锁的未释放问题

- 内存分配器通常使用内部锁（例如 pthread\_mutex）保证线程安全。如果父进程在 fork 前持有分配器的锁，子进程会继承这个锁的锁定状态，但无法感知它已被复制。这可能导致子进程尝试获取一个“已锁定”的锁，从而死锁。

## 3.4 q4

### ? Question

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] ([https://en.cppreference.com/w/cpp/atomic/memory\\_order](https://en.cppreference.com/w/cpp/atomic/memory_order)) 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。

`Ordering` 枚举指定了操作对内存可见性和同步性要求，其语义与 C++20 中的 `memory_order` 概念一致。主要的枚举值包括：

- **Relaxed** 只保证原子性，不提供任何额外的同步或排序保证；适合只需确保读写不会产生数据破坏，但不需要跨线程排序的场景。
- **Acquire** 用于加载操作，确保在该原子操作之后的所有读写不能被重排到该操作之前。其他线程使用 `Release` 存储写入的值，对当前线程来说是“获取”了先前的写入效果。
- **Release** 用于存储操作，确保在该原子操作之前的所有读写都完成后，才允许该操作向其他线程“发布”。当其他线程执行 `Acquire` 读取后，可以看到 `Release` 之前的所有操作效果。
- **AcqRel (AcquireRelease)** 同时具备 `Acquire` 和 `Release` 效果，常用于读-改-写操作，确保既能发布之前的写入，也能获取之后的读取。
- **SeqCst (Sequentially Consistent)** 最强的同步保证，实现全局顺序一致性：所有的 `SeqCst` 原子操作在一个全局统一的次序下执行。这个选项使得代码易于理解和调试，但可能带来一定性能开销。

## 3.5 q5

### ? Question

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？

- **Sync**：一个类型实现了 `Sync` 意味着它的引用（即 `&T`）可以安全地在多个线程之间共享。对于像 `SpinLock` 这样的锁，其本质作用就是在多个线程间同步对数据的访问，因此必须保证可以通过共享引用安全地访问，否则编译器会阻止在跨线程共享时出现数据竞争的不安全操作。
- **Send**：一个类型实现了 `Send` 表示该类型的所有权可以安全地从一个线程传递到另一个线程（例如通过线程的启动参数）。这保证了在跨线程传递对象时，不会引入未定义行为或数据竞争。



在实现 `SpinLock` 时，通常会包含一个用于表示锁状态的原子变量。如果这个类型没有手动实现（或自动推导出）`Sync`，那么使用该锁来保护共享数据时就存在安全隐患，因为编译器会认为其在多线程共享时不安全。同理，如果 `SpinLock` 没有实现 `Send`，就无法在线程之间传递，限制了其在并发场景下的使用。

因此，为了让 `SpinLock` 既能在不同线程间共享（通过共享引用）又能跨线程传递（通过所有权转移），需要确保它同时实现了 `Sync` 和 `Send`。

## 3.6 q6

### ? Question

`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？

`core::hint::spin_loop` 内部通常使用 CPU 的 `PAUSE` 指令，它仅提供一个忙等待（busy-wait）时的小幅延迟和资源缓解提示，告诉处理器“稍候检查”，而不会停机或降低线程的活跃度。相比之下，`hlt` 指令会让 CPU 进入休眠状态，直到下一个外部中断唤醒它。不能使用 `hlt` 的原因有几点：

#### 1. 适用场景不同

- `spin_loop` 适合忙等待或自旋锁场景，线程需要持续轮询检查某个状态，而 `PAUSE` 指令可降低功耗和改善超线程性能。
- `hlt` 用于让 CPU 处于空闲状态，但它会停止当前核心的执行，并依赖中断唤醒，这不适合需要频繁检查状态变化的自旋场景。

2. 特权级限制 `hlt` 是特权指令，只能在内核态执行；在用户态或某些非内核代码中调用可能会触发保护异常。而 `spin_loop` 则可以安全用于用户态或内核态的忙等待中。

3. 响应性要求 自旋锁通常要求线程持续快速地轮询资源状态，使用 `hlt` 会导致线程在等待期间长时间挂起，反而降低响应性。

## 4 加分项

### 4.1 fish

