中山大學

SUN YAT-SEN UNIVERSITY

实验报告

实验五：进程复制的实现、并发与锁机制

姓　　名	元朗曦
学　　号	23336294
班　　级	计算机八班
专　　业	计算机科学与技术
学　　院	计算机学院

1 fork 的实现

YSOS 的 fork 系统调用设计如下描述：

- fork 会创建一个新的进程，新进程称为子进程，原进程称为父进程。
- 子进程在系统调用后将得到 0 的返回值，而父进程将得到子进程的 PID。 如果创建失败，父进程将得到 -1 的返回值。
- fork 不复制父进程的内存空间，不实现 CoW (Copy on Write) 机制，即父子进程将持有一定的共享内存：代码段、数据段、堆、bss 段等。
- fork 子进程与父进程共享内存空间（页表），但子进程拥有自己独立的寄存器和栈空间，即在一个不同的栈的地址继承原来的数据。
- 由于上述内存分配机制的限制，fork 系统调用必须在任何 Rust 内存分配（堆内存分配）之前进行。

1.1 系统调用

首先编辑 pkg/syscall/src/lib.rs，添加 fork 系统调用号：

```rust
#[repr(usize)]
#[derive(Clone, Debug, FromPrimitive)]
pub enum Syscall {
    // ...

    Fork = 58,

    // ...
}
```

然后在 pkg/kernel/src/interrupt/syscall/{mod,service}.rs 中添加相关实现：

```rust
 pub fn dispatcher(context: &mut ProcessContext) {
     // ...

     match args.syscall {
         // ...

         // None -> pid: u16 or 0 or -1
         Syscall::Fork => sys_fork(context),
     }
 }
```

```rust
pub fn sys_fork(context: &mut ProcessContext) {
    fork(context)
}
```

1.2 进程管理

我们将 fork 的功能拆分，逐层委派给下一级，在 proc 模块中实现。编辑 pkg/kernel/src/proc/{mod,manager,process,vm/mod,paging,vm/stack}.rs 如下：

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

```rust
impl ProcessManager {
    // ...

    pub fn fork(&self) {
        // DONE: get current process
        // DONE: fork to get child
        // DONE: add child to process list
        let proc = self.current().fork();
        let pid = proc.pid();
        self.add_proc(pid, proc);
        self.push_ready(pid);

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

```rust
impl Process {
    // ...

    pub fn fork(self: &Arc<Self>) -> Arc<Self> {
        // DONE: lock inner as write
        // DONE: inner fork with parent weak ref
        let mut inner = self.write();
        let child_inner = inner.fork(Arc::downgrade(self));
        let child_pid = ProcessId::new();

        // FOR DBG: maybe print the child process info?
        debug!("{}#{} forked from {}#{}", child_inner.name(), child_pid,
        inner.name(), self.pid);

        // DONE: make the arc of child
```

```
        // DONE: add child to current process's children list
        // DONE: set fork ret value for parent with `context.set_rax`
        // DONE: mark the child as ready & return it
        let child = Arc::new(Self {
            pid: child_pid,
            inner: Arc::new(RwLock::new(child_inner)),
        });
        inner.children.push(child.clone());
        inner.context.set_rax(child_pid.0 as usize);
        inner.pause();
        child
    }
}

impl ProcessInner {
    // ...

    pub fn fork(&mut self, parent: Weak<Process>) -> ProcessInner {
        // DONE: fork the process virtual memory struct
        // DONE: calculate the real stack offset
        // DONE: update `rsp` in interrupt stack frame
        // DONE: set the return value 0 for child with `context.set_rax`
        let new_vm = self.vm().fork(self.children.len() as u64 + 1u64);
        let offset = new_vm.stack.stack_offset(&self.vm().stack);

        let mut new_context = self.context;
        new_context.set_stack_offset(offset);
        new_context.set_rax(0);

        // DONE: clone the process data struct
        // DONE: construct the child process inner
        Self {
            name: self.name.clone(),
            exit_code: None,
            parent: Some(parent),
            status: ProgramStatus::Ready,
            ticks_passed: 0,
            context: new_context,
            children: Vec::new(),
            proc_vm: Some(new_vm),
            proc_data: self.proc_data.clone(),
        }
        // NOTE: return inner because there's no pid record in inner
    }
}
```

```rust
impl ProcessVm {
    // ...

    pub fn fork(&self, stack_offset_count: u64) -> Self {
        // clone the page table context (see instructions)
        let new_page_table = self.page_table.fork();

        let mapper = &mut new_page_table.mapper();
        let alloc = &mut *get_frame_alloc_for_sure();

        Self {
            page_table: new_page_table,
            stack: self.stack.fork(mapper, alloc, stack_offset_count),
        }
    }
}
```

```rust
impl PageTableContext {
    // ...

    pub fn fork(&self) -> Self {
        // forked process shares the page table
        Self {
            reg: self.reg.clone(),
        }
    }
}
```

```rust
impl Stack {
    // ...

    pub fn fork(
        &self,
        mapper: MapperRef,
        alloc: FrameAllocatorRef,
        stack_offset_count: u64,
    ) -> Self {
        // DONE: alloc & map new stack for child (see instructions)
        // DONE: copy the *entire stack* from parent to child
        let cur_stack_base = self.range.start.start_address().as_u64();
        let mut new_stack_base = cur_stack_base - stack_offset_count * STACK_MAX_SIZE;

        while elf::map_pages(new_stack_base, self.usage, mapper, alloc, true).is_err() {
            trace!("Mapping thread stack to {:#x} failed.", new_stack_base);
            new_stack_base -= STACK_MAX_SIZE;
```

```
        }
        debug!("Mapping thread stack to {:#x} succeeded.", new_stack_base);

        self.clone_range(cur_stack_base, new_stack_base, self.usage);

        // DONE: return the new stack
        let new_start = Page::containing_address(VirtAddr::new(new_stack_base));
        Self {
            range: Page::range(new_start, new_start + self.usage),
            usage: self.usage,
        }
    }
}
```

1.3 功能测试

创建 `pkg/app/forktest` 包，修改 `main.rs` 内容如下：

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

extern crate alloc;
extern crate lib;

use lib::*;

static mut M: u64 = 0xdeadbeef;

fn main() -> isize {
    let mut c = 32;
    let m_ptr = &raw mut M;

    // do not alloc heap before `fork`
    // which may cause unexpected behavior since we won't copy the heap in
    `fork`
    let pid = sys_fork();

    if pid == 0 {
        println!("I am the child process");

        assert_eq!(c, 32);

        unsafe {
            println!("child read value of M: {:#x}", *m_ptr);
            *m_ptr = 0x2333;
            println!("child changed the value of M: {:#x}", *m_ptr);
        }
```

```
29
30          c += 32;
31      } else {
32          println!("I am the parent process");
33
34          sys_stat();
35
36          assert_eq!(c, 32);
37
38          println!("Waiting for child to exit...");
39
40          let ret = sys_wait_pid(pid);
41
42          println!("Child exited with status {}", ret);
43
44          assert_eq!(ret, 64);
45
46          unsafe {
47              println!("parent read value of M: {:#x}", *m_ptr);
48              assert_eq!(*m_ptr, 0x2333);
49          }
50
51          c += 1024;
52
53          assert_eq!(c, 1056);
54      }
55
56      c
57  }
58
59  entry!(main);
```

编译并运行我们的操作系统，在 Shell 中输入 `exec forktest`，得到如下输出：

```
1   [D] Spawned process: forktest#3
2   [D] Mapping thread stack to 0x3ffefffff000 succeeded.
3   [D] forktest#4 forked from forktest#3
4   [D] Ready Queue: [2, 1, 3, 4]
5   I am the parent process
6     PID | PPID | Process Name |  Ticks   | Status
7    #  1 | #  0 | kernel       |   12490 | Ready
8    #  2 | #  1 | sh           |   12481 | Ready
9    #  3 | #  2 | forktest     |       5 | Running
10   #  4 | #  3 | forktest     |       2 | Ready
11  Queue  : [4, 3, 1]
12  CPUs   : [0: 3]
```

```
I am the child process
Waiting for child to exit...
child read value of M: 0xdeadbeef
child changed the value of M: 0x2333
[D] Killing process forktest#4 with ret code: 64
Child exited with status 64
parent read value of M: 0x2333
[D] Killing process forktest#3 with ret code: 1056
```

可以看到，父子进程共享内存空间，父进程在子进程退出后能读取到子进程修改的变量值。

2 进程的阻塞与唤醒

在先前的实现中，我们已经实现了 `wait_pid` 系统调用，它通过轮询的方式来等待一个进程的退出，并返回其退出状态。轮询会消耗大量的 CPU 时间，因此我们需要一种更为高效的方式来进行进程的阻塞与唤醒。

2.1 等待队列

在 `pkg/kernel/src/proc/manager.rs` 中，修改 `ProcessManager` 并添加等待队列：

```rust
pub struct ProcessManager {
    // ...

    wait_queue: Mutex<BTreeMap<ProcessId, BTreeSet<ProcessId>>>,
}
```

其中，`BTreeMap` 的键值为被等待的进程的编号，`BTreeSet` 为等待中进程编号的集合。

2.2 阻塞进程

为 `ProcessManager` 添加 `block` 函数，用于将进程设置为阻塞状态：

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

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）：

```rust
pub fn wait_pid(pid: ProcessId, context: &mut ProcessContext) {
    x86_64::instructions::interrupts::without_interrupts(|| {
        let manager = get_process_manager();
        if let Some(ret) = manager.get_exit_code(pid) {
            context.set_rax(ret as usize);
        } else {
            manager.wait_pid(pid);
```

```
                manager.save_current(context);
                manager.current().write().block();
                manager.switch_next(context);
            }
        })
    }
```

同时为 `ProcessManager` 添加 `wait_pid` 函数：

```Rust
pub fn wait_pid(&self, pid: ProcessId) {
    let mut wait_queue = self.wait_queue.lock();
    // DONE: push the current process to the wait queue
    let entry = wait_queue.entry(pid).or_default();
    entry.insert(processor::get_pid());
}
```

2.3 唤醒进程

在阻塞进程后，还需要对进程进行唤醒。对于本处的 `wait_pid` 系统调用，当被等待的进程退出时，需要唤醒等待队列中的进程。

首先，为 `ProcessManager` 添加 `wake_up` 函数：

```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();
        if let Some(ret) = ret {
            // DONE: set the return value of the process
            inner.set_return(ret as usize);
        }
        // DONE: set the process as ready
        // DONE: push to ready queue
        inner.pause();
        self.push_ready(pid);
    }
}
```

在进程退出时，也即 `kill` 系统调用中，需要唤醒等待队列中的进程。修改 `ProcessManager` 中的 `kill` 函数：

```Rust
pub fn kill(&self, pid: ProcessId, ret: isize) {
    let Some(proc) = self.get_proc(&pid) else {
        error!("Process #{} not found.", pid);
        return;
    };

```

```
    if proc.read().is_dead() {
        error!("Process #{} is already dead.", pid);
        return;
    }

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

    proc.kill(ret);
}
```

这样，就实现了一个无需轮询的进程阻塞与唤醒机制。

2.4 功能测试

我们可以在 Shell 用 `exec sh` 创建另一个 Shell，然后执行 `list proc`，得到如下输出：

```
  PID | PPID | Process Name |  Ticks   | Status
  #  1 | #  0 | kernel       |   33251 | Ready
  #  2 | #  1 | sh           |   22040 | Blocked
  #  3 | #  2 | sh           |   11204 | Running
Queue  : [1]
CPUs   : [0: 3]
```

可以看到，第一个 Shell 被成功阻塞。退出第二个 Shell 后再执行 `list proc`，得到如下输出：

```
  PID | PPID | Process Name |  Ticks   | Status
  #  1 | #  0 | kernel       |  335202 | Ready
  #  2 | #  1 | sh           |   32732 | Running
Queue  : [1]
CPUs   : [0: 2]
```

可以看到，第一个 Shell 被成功唤醒。

3 并发与锁机制

由于并发执行时，线程的调度顺序无法预知，进而造成的执行顺序不确定，持有共享资源的进程之间的并发执行可能会导致数据的不一致，最终导致相同的程序产生一系列不同的结果，这样的情况被称之为竞态条件。

3.1 原子指令

一般而言，为了解决并发任务带来的问题，需要通过指令集中的原子操作来保证数据的一致性。在 Rust 中，这类原子指令被封装在 `core::sync::atomic` 模块中，作为架构无关的原子操作来提供并发安全性。

我们可以利用原子指令来为用户态程序提供两种简单的同步操作：自旋锁 `SpinLock` 和信号量 `Semaphore`。其中自旋锁的实现并不需要内核态的支持，而信号量则会涉及到进程调度等操作，需要内核态的支持。

3.2 自旋锁

自旋锁 SpinLock 是一种简单的锁机制，它通过不断地检查锁的状态来实现线程的阻塞，直到获取到锁为止。

创建 pkg/lib/src/sync.rs，完成 SpinLock 的基础实现：

```rust
pub struct SpinLock {
    bolt: AtomicBool,
}

impl SpinLock {
    pub const fn new() -> Self {
        Self {
            bolt: AtomicBool::new(false),
        }
    }

    fn try_acquire(&mut self) -> bool {
        self.bolt
            .compare_exchange(
                false,
                true,
                Ordering::Acquire,
                Ordering::Relaxed,
            )
            .is_ok()
    }
    pub fn acquire(&mut self) {
        // DONE: acquire the lock, spin if the lock is not available
        while ! self.try_acquire() { core::hint::spin_loop(); }
    }

    pub fn release(&mut self) {
        // DONE: release the lock
        self.bolt.store(false, Ordering::Relaxed);
    }
}

impl Default for SpinLock {
    fn default() -> Self {
        Self::new()
    }
}

unsafe impl Sync for SpinLock {}
```

3.3 信号量

得利于 Rust 良好的底层封装，自旋锁的实现非常简单。但是也存在一定的问题：

- 忙等待：自旋锁会一直占用 CPU 时间，直到获取到锁为止，这会导致 CPU 利用率的下降。
- 饥饿：如果一个线程一直占用锁，其他线程可能会一直无法获取到锁。
- 死锁：如果两个线程互相等待对方占有的锁，就会导致死锁。

信号量 `Semaphore` 是一种更为复杂的同步机制，它可以用于控制对共享资源的访问，也可以用于控制对临界区的访问。通过与进程调度相关的操作，信号量还可以用于控制进程的执行顺序、提高 CPU 利用率等。

在 `pkg/kernel/src/proc/sync.rs` 中添加 `Semaphore` 的相关实现：

```rust
#[derive(Clone, Copy, Debug, Eq, PartialEq, Ord, PartialOrd)]
pub struct SemaphoreId(u32);

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

/// Mutex is required for Semaphore
#[derive(Debug, Clone)]
pub struct Semaphore {
    count: usize,
    wait_queue: VecDeque<ProcessId>,
}

/// Semaphore result
#[derive(Debug)]
pub enum SemaphoreResult {
    Ok,
    NotExist,
    Block(ProcessId),
    WakeUp(ProcessId),
}

impl Semaphore {
    /// Create a new semaphore
    pub fn new(value: usize) -> Self {
        Self {
            count: value,
            wait_queue: VecDeque::new(),
        }
    }

```

```rust
    /// Wait the semaphore (acquire/down/proberen)
    ///
    /// if the count is 0, then push the process into the wait queue
    /// else decrease the count and return Ok
    pub fn wait(&mut self, pid: ProcessId) -> SemaphoreResult {
        // DONE: if the count is 0, then push pid into the wait queue, return Block(pid)
        // DONE: else decrease the count and return Ok
        if self.count == 0 {
            self.wait_queue.push_back(pid);
            SemaphoreResult::Block(pid)
        } else {
            self.count -= 1;
            SemaphoreResult::Ok
        }
    }

    /// Signal the semaphore (release/up/verhogen)
    ///
    /// if the wait queue is not empty, then pop a process from the wait queue
    /// else increase the count
    pub fn signal(&mut self) -> SemaphoreResult {
        // DONE: if the wait queue is not empty, pop a process from the wait queue, return WakeUp(pid)
        // DONE: else increase the count and return Ok
        if let Some(pid) = self.wait_queue.pop_front() {
            SemaphoreResult::WakeUp(pid)
        } else {
            self.count += 1;
            SemaphoreResult::Ok
        }
    }
}

#[derive(Debug, Default)]
pub struct SemaphoreSet {
    sems: BTreeMap<SemaphoreId, Mutex<Semaphore>>,
}

impl SemaphoreSet {
    pub fn insert(&mut self, key: u32, value: usize) -> bool {
        trace!("Sem Insert: <{:#x}>{}", key, value);

        // DONE: insert a new semaphore into the sems, use `insert(/* ... */).is_none()`
```

```
77              self.sems.insert(SemaphoreId::new(key),
                Mutex::new(Semaphore::new(value))).is_none()
78          }
79
80          pub fn remove(&mut self, key: u32) -> bool {
81              trace!("Sem Remove: <{:#x}>", key);
82
83              // DONE: remove the semaphore from the sems, use `remove(/* ...
                */).is_some()`
84              self.sems.remove(&SemaphoreId::new(key)).is_some()
85          }
86
87          /// Wait the semaphore (acquire/down/proberen)
88          pub fn wait(&self, key: u32, pid: ProcessId) -> SemaphoreResult {
89              let sid = SemaphoreId::new(key);
90
91              // DONE: try get the semaphore from the sems, then do it's operation
92              // DONE: return NotExist if the semaphore is not exist
93              if let Some(sem) = self.sems.get(&sid) {
94                  let mut locked = sem.lock();
95                  trace!("Sem Wait  : <{:#x}>{}", key, locked);
96                  locked.wait(pid)
97              } else {
98                  SemaphoreResult::NotExist
99              }
100         }
101
102         /// Signal the semaphore (release/up/verhogen)
103         pub fn signal(&self, key: u32) -> SemaphoreResult {
104             let sid = SemaphoreId::new(key);
105
106             // DONE: try get the semaphore from the sems, then do it's operation
107             // DONE: return NotExist if the semaphore is not exist
108             if let Some(sem) = self.sems.get(&sid) {
109                 let mut locked = sem.lock();
110                 trace!("Sem Signal: <{:#x}>{}", key, locked);
111                 locked.signal()
112             } else {
113                 SemaphoreResult::NotExist
114             }
115         }
116 }
117
118 impl core::fmt::Display for Semaphore {
119     fn fmt(&self, f: &mut core::fmt::Formatter<'_>) -> core::fmt::Result {
120         write!(f, "Semaphore({}) {:?}", self.count, self.wait_queue)
```

```
121     }
122 }
```

其中，SemaphoreId 用于标识信号量；Semaphore 利用一个 usize 和 VecDeque 记录该信号量的需求情况，并实现一些基本操作；SemaphoreSet 用于管理信号量集合。

为 enum Syscall 添加信号量相关的系统调用号：

```rust
Sem = 66,
```

然后在 pkg/kernel/src/interrupt/syscall/{mod,service}.rs 中添加信号量相关的系统调用实现：

```rust
pub fn dispatcher(context: &mut ProcessContext) {
    // ...

    match args.syscall {
        // ...

        // op: u8, key: u32, val: usize -> ret: any
        Syscall::Sem => sys_sem(&args, context),
    }
}
```

```rust
pub fn sys_sem(args: &SyscallArgs, context: &mut ProcessContext) {
    match args.arg0 {
        0 => context.set_rax(new_sem(args.arg1 as u32, args.arg2)),
        1 => context.set_rax(remove_sem(args.arg1 as u32)),
        2 => sem_signal(args.arg1 as u32, context),
        3 => sem_wait(args.arg1 as u32, context),
        _ => context.set_rax(usize::MAX),
    }
}
```

这之后与实现 fork 时类似，将功能逐层委派。在 pkg/kernel/src/proc/{mod,data}.rs 中添加相关实现：

```rust
pub fn new_sem(key: u32, value: usize) -> usize {
    x86_64::instructions::interrupts::without_interrupts(|| {
        if get_process_manager().current().write().new_sem(key, value) {
            0
        } else {
            1
        }
    })
}
pub fn remove_sem(key: u32) -> usize {
    x86_64::instructions::interrupts::without_interrupts(|| {
        if get_process_manager().current().write().remove_sem(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 ret = manager.current().write().sem_signal(key);
        match ret {
            SemaphoreResult::Ok => context.set_rax(0),
            SemaphoreResult::NotExist => context.set_rax(1),
            SemaphoreResult::WakeUp(pid) => manager.wake_up(pid, None),
            _ => unreachable!(),
        }
    })
}
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),
            SemaphoreResult::Block(pid) => {
                // DONE: save, block it, then switch to next
                manager.save_current(context);
                manager.block(pid);
                manager.switch_next(context);
            }
            _ => unreachable!(),
        }
    })
}
```

```Rust
#[derive(Debug, Clone)]
pub struct ProcessData {
    // ...

    pub(super) semaphores: Arc<RwLock<SemaphoreSet>>,
}

// ...

```

```rust
impl ProcessData {
    // ...

    #[inline]
    pub fn new_sem(&mut self, key: u32, value: usize) -> bool {
        self.semaphores.write().insert(key, value)
    }
    #[inline]
    pub fn remove_sem(&mut self, key: u32) -> bool {
        self.semaphores.write().remove(key)
    }
    #[inline]
    pub fn sem_signal(&mut self, key: u32) -> SemaphoreResult {
        self.semaphores.read().signal(key)
    }
    #[inline]
    pub fn sem_wait(&mut self, key: u32, pid: ProcessId) -> SemaphoreResult {
        self.semaphores.read().wait(key, pid)
    }
}
```

3.4 测试任务

3.4.1 多线程计数器

在 pkg/app/counter 包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。具体实现如下：

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

use lib::*;
extern crate lib;

const THREAD_COUNT: usize = 8;
static mut COUNTER: isize = 0;

static MUTEX: Semaphore = Semaphore::new(0xDEADBEEF);

fn main() -> isize {
    MUTEX.init(1);
    let mut pids = [0u16; THREAD_COUNT];

    for i in 0..THREAD_COUNT {
        let pid = sys_fork();
        if pid == 0 {
            do_counter_inc();
            sys_exit(0);
```

```
        } else {
            pids[i] = pid; // only parent knows child's pid
        }
    }

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

    for i in 0..THREAD_COUNT {
        println!("#{} waiting for #{}...", cpid, pids[i]);
        sys_wait_pid(pids[i]);
    }

    println!("COUNTER result: {}", unsafe { COUNTER });

    0
}

fn do_counter_inc() {
    for _ in 0..100 {
        // DONE: protect the critical section
        MUTEX.wait();
        inc_counter();
        MUTEX.signal();
    }
}

/// Increment the counter
///
/// this function simulate a critical section by delay
/// DO NOT MODIFY THIS FUNCTION
fn inc_counter() {
    unsafe {
        delay();
        let mut val = COUNTER;
        delay();
        val += 1;
        delay();
        COUNTER = val;
    }
}

#[inline(never)]
#[unsafe(no_mangle)]
fn delay() {
```

```
    for _ in 0..0x100 {
        core::hint::spin_loop();
    }
}

entry!(main);
```

计数器的值最终应为 800。

3.4.2 消息队列

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：

- 父进程使用 fork 创建额外的 16 个进程，其中一半为生产者，一半为消费者。
- 生产者不断地向消息队列中写入消息，消费者不断地从消息队列中读取消息。
- 每个线程处理的消息总量共 10 条。
- 即生产者会产生 10 个消息，每个消费者只消费 10 个消息。
- 在每个线程生产或消费的时候，输出相关的信息。
- 你可能需要使用信号量或旋锁来实现一个互斥锁，保证操作和信息输出之间不会被打断。
- 在生产者和消费者完成上述操作后，使用 sys_exit(0) 直接退出。
- 最终使用父进程等待全部的子进程退出后，输出消息队列的消息数量。
- 在父进程创建完成 16 个进程后，使用 sys_stat 输出当前的全部进程的信息。

具体实现如下：

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

use lib::*;
extern crate lib;

static MUTEX: Semaphore = Semaphore::new(0xBABEBABE);
static EMPTY: Semaphore = Semaphore::new(0xBABFBABF);

static mut COUNT: usize = 0;

entry!(main);
fn main() -> isize {
    MUTEX.init(1);
    EMPTY.init(0);

    let mut pids = [0u16; 16];
    // Fork producers and consumers.
    for i in 0..16 {
        let pid = sys_fork();
        if pid == 0 { // Child Branch
```

```rust
            if i % 2 == 0 { producer() } else { consumer() }
        } else { // Parent Branch
            pids[i] = pid;
        }
    }

    // Print information of current processes.
    sys_stat();

    // Wait for all children to exit.
    for pid in pids {
        println!("Waiting for child process #{}", pid);
        sys_wait_pid(pid);
    }

    MUTEX.free();
    EMPTY.free();

    0
}

fn producer() -> ! {
    let pid = sys_get_pid();
    for _ in 0..10 {
        delay();
        // Wait for other IO operations.
        MUTEX.wait();
        // Add a message (simulated by a number).
        unsafe {
            COUNT += 1;
        }
        println!("Process #{pid} produced a message, current count: {}", unsafe { COUNT });
        // Signal on finishing.
        MUTEX.signal();
        // Signal that the queue is not empty.
        EMPTY.signal();
    }
    sys_exit(0);
}

fn consumer() -> ! {
    let pid = sys_get_pid();
    for _ in 0..10 {
        delay();
        // Wait if message queue is empty.
```

```
67          EMPTY.wait();
68          // Wait for other IO operations.
69          MUTEX.wait();
70          // Remove a message (simulated by a number).
71          unsafe {
72              COUNT -= 1;
73          }
74          println!("Process #{pid} consumed a message, current count: {}", unsafe
            { COUNT });
75          // Signal on finishing.
76          MUTEX.signal();
77      }
78      sys_exit(0);
79  }
80
81  #[inline(never)]
82  #[unsafe(no_mangle)]
83  fn delay() {
84      for _ in 0..0x100 {
85          core::hint::spin_loop();
86      }
87  }
```

> 🔥 **注意**
>
> 我们并不需要真的实现一个消息队列，因为我们只关心消息队列的大小，用一个变量来记录即可。

3.4.3 哲学家的晚饭

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。

创建一个用户程序 `pkg/app/dinner`，实现并解决哲学家就餐问题：

- 创建一个程序，模拟五个哲学家的行为。
- 每个哲学家都是一个独立的线程，可以同时进行思考和就餐。
- 使用互斥锁来保护每个筷子，确保同一时间只有一个哲学家可以拿起一根筷子。
- 使用等待操作调整哲学家的思考和就餐时间，以增加并发性和实际性。
- 当哲学家成功就餐时，输出相关信息，如哲学家编号、就餐时间等。

- 向程序中引入一些随机性，例如在尝试拿筷子时引入一定的延迟，模拟竞争条件和资源争用。
- 可以设置等待时间或循环次数，以确保程序能够运行足够长的时间，并尝试观察到不同的情况，如死锁和饥饿。

具体实现如下：

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

use lib::*;
extern crate lib;

static MUTEX: Semaphore = Semaphore::new(0xBADBABE);
static CHOPSTICKS: [Semaphore; 5] = semaphore_array![ 0, 1, 2, 3, 4 ];

entry!(main);
fn main() -> isize {
    MUTEX.init(4);
    for i in 0..5 {
        CHOPSTICKS[i].init(1);
    }

    let mut pids = [0u16; 5];
    // Fork philosophers.
    for i in 0..5 {
        let pid = sys_fork();
        if pid == 0 { // Child Branch
            philosopher(i);
        } else { // Parent Branch
            pids[i] = pid;
        }
    }

    sys_stat();

    for pid in pids {
        println!("Waiting for child process #{}", pid);
        sys_wait_pid(pid);
    }

    MUTEX.free();
    for i in 0..5 {
        CHOPSTICKS[i].free();
    }

    0
```

```rust
}

fn philosopher(id: usize) -> ! {
    let pid = sys_get_pid();

    for _ in 0..0x100 {
        // Think
        println!("Philosopher #{id} (process #{pid}) is thinking");
        delay();

        // Eat
        MUTEX.wait();
        CHOPSTICKS[(id + 0) % 5].wait();
        CHOPSTICKS[(id + 1) % 5].wait();
        println!("Philosopher #{id} (process #{pid}) is eating");
        CHOPSTICKS[(id + 0) % 5].signal();
        CHOPSTICKS[(id + 1) % 5].signal();
        MUTEX.signal();
    }

    sys_exit(0);
}

#[inline(never)]
#[unsafe(no_mangle)]
fn delay() {
    for _ in 0..0x100 {
        core::hint::spin_loop();
    }
}
```

4 思考题

1. 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。

解答

当使用 Mutex 保护同步队列时，可能会出现以下问题：

1. 死锁问题：

   如果在 pop 操作过程中持有 Mutex 锁时，发生串口输入中断，而中断处理程序也尝试获取相同的 Mutex 锁，就会导致死锁。

2. 优先级反转：

   如果高优先级的中断处理程序需要等待低优先级的线程释放 Mutex，可能会导致优先级反转问题，影响系统的实时性。

3. 中断上下文操作受限：

   中断处理程序通常运行在中断上下文中，不能进行阻塞操作，否则会影响系统的中断响应能力。

为了解决上述问题，可以在进入临界区前禁用中断，避免中断处理程序抢占 CPU 并尝试获取 Mutex 锁。具体地，在主线程中调用 pop 时，先禁用中断，然后获取 Mutex 锁；在操作完成后，释放 Mutex 锁并启用中断；中断处理程序无需获取锁，直接操作队列。

2. 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？

解答

1. 父进程页表：
   - 父进程的页表记录了虚拟地址空间与物理内存的映射关系，包括代码段、数据段、堆、bss 段等。
   - 在 YSOS 的 `fork` 中，父子进程共享这些段的页表，因此不需要为子进程复制这些页表项。
2. 子进程页表：
   - 子进程的页表起初与父进程一致，共享代码段、数据段、堆和 bss 段的映射。
   - 子进程拥有独立的栈空间，因此需要为子进程的栈单独分配内存，并在子进程的页表中建立独立的映射。
3. 共享页表的影响：
   - 父子进程共享内存的页表（代码段、数据段等），意味着这些内存是直接共享的，修改会实时反映到对方的进程中。
   - 这种设计避免了 CoW 的复杂性，但需要程序员保证父子进程之间对共享内存的访问不会冲突。
4. 内核页表：
   - 父子进程共享内核页表，用于映射内核空间的地址。
   - 这部分不受 `fork` 的影响。

需要注意的问题有

1. 栈的独立性
   - 子进程栈的分配：
     - 在 `fork` 过程中，需要为子进程分配新的栈空间，并复制父进程栈的内容。
     - 子进程的寄存器指向独立的栈地址，确保父子进程的栈操作互不干扰。
   - 栈内容的复制：
     - 必须在 fork 过程中将父进程栈的内容复制到子进程的栈中。
     - 需要注意，栈中可能包含指针，复制时必须保证这些指针仍然指向共享的内存地址，而不能错误地指向子进程的栈。
2. 共享内存的管理
   - 同步问题：
     - 由于父子进程共享代码段、数据段、堆和 bss 段，任何一方对这些段的修改都会影响另一方。
     - 程序员需要明确地管理访问，避免父子进程同时修改共享内存，导致数据不一致或冲突。
   - 数据一致性：
     - 如果父子进程需要独立的数据段或堆（例如为了各自维护状态），则需要手动

3. 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？

> **解答**
>
> 在实验的实现中，`fork` 系统调用必须在任何 Rust 的堆内存分配之前进行，主要是因为实验中的 `fork` 系统调用具有以下特殊机制：
>
> 1. 父子进程共享内存：
>   - 在 YSOS 的 `fork` 实现中，父子进程共享内存空间（包括代码段、数据段、堆、bss 段等）。
>   - 由于没有实现 COW（Copy-on-Write）机制，父子进程对共享内存的修改会直接影响另一方。
> 2. 内存分配器的状态共享：
>   - Rust 的堆内存分配器（如 `alloc` 或其他内存分配库）维护着全局的分配状态（如空闲块、已分配块等）。
>   - 如果在 `fork` 之后，父子进程共享堆内存空间，则两个进程的内存分配器会同时操作同一个全局状态。
>   - 这可能导致分配器的状态被破坏，从而引发未定义行为，包括内存泄漏、分配失败、内存访问冲突等问题。
>
> 如果在堆内存分配之后进行 `fork`，可能会出现的问题
>
> 1. 内存分配器状态的破坏：
>   - 堆内存分配器通常依赖于元数据（如内存块的大小、空闲链表等）来跟踪分配和释放的内存。
>   - 如果父进程在 `fork` 之前分配了一些内存，`fork` 之后父子进程共享这些元数据，但彼此独立释放或分配内存，则分配器的元数据会被破坏。
> 2. 数据不一致性：
>   - 如果父子进程同时操作共享堆内存，可能会导致数据不一致。
> 3. 内存泄漏：
>   - 如果子进程分配了内存但没有释放，而父进程并不知道这些分配情况，可能会导致内存泄漏。由于分配器的状态是共享的，但子进程的生命周期可能短于父进程，子进程的分配行为可能影响系统的长期内存使用。
> 4. 竞争条件：
>   - 在多线程程序中，如果父进程的线程在 `fork` 之前进行了堆内存的分配操作，那么 `fork` 后子进程可能只继承了父进程的一个线程。此时，分配器的状态可能处于不一致的中间状态，导致未定义行为。
> 5. 未定义行为：
>   - 如果内存分配器的实现依赖于某些未被 `fork` 机制正确处理的底层特性（如锁、线程局部存储等），则在 `fork` 之后，分配器的行为可能变得不可预测。

4. 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。

> **解答**
>
> 1. `Relaxed`
>   - 不对其他线程的内存操作进行排序限制。
> 2. `Acquire`
>   - 保证当前线程在执行此操作后，能看到其他线程在此操作之前所有的写入行为。
> 3. `Release`
>   - 保证当前线程在执行此操作之前的所有写入行为，对其他线程可见。
> 4. `AcqRel`
>   - 同时具有 `Acquire` 和 `Release` 的语义。
> 5. `SeqCst`
>   - 强一致性顺序，所有原子操作都按照全局一致的顺序执行。

5. 在实现 `SpinLock` 的时候，为什么需要实现 `Sync` 特性？类似的 `Send` 特性又是什么含义？

> **解答**
>
> - `Sync` 特性使它可以安全地在多个线程中共享引用；
> - `Send` 特性使它可以安全地在不同线程之间传递所有权。

6. `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？

> **解答**
>
> - `pause` 指令用于自旋等待，允许 CPU 进入低功耗状态，减少功耗和热量。
> - `hlt` 指令会将 CPU 置于休眠状态，直到下一个中断到来，这会导致 CPU 停止执行当前线程的代码。
>
> 在自旋锁的实现中，使用 `hlt` 会导致 CPU 停止执行当前线程的代码，从而无法响应其他线程的请求，因此不能使用 `hlt` 指令。