Lab8

编程题：

1 ** 在 Linux 环境下，请用互斥锁和条件变量实现哲学家就餐的多线程应用程序。

答：想要实现这个任务，我们需要模拟场景，即经典的“哲学家就餐问题”（Dining Philosophers），使用互斥锁模拟叉子，验证无死锁且有人能正常进餐。这里我们可以使用的无死锁策略：统一“拿锁顺序”（先拿编号小的叉子）。对于效果评估，我们可以依靠时间戳和字符图展示各哲学家的“思考-等待-进餐”状态随时间变化，直观分析并发性能。

根据以上，我们可以编写代码如下：

```rust
const N: usize = 5;// 哲学家数量
const ROUND: usize = 4; // 每个哲学家思考/进餐轮数
// A round: think -> wait for forks -> eat
const GRAPH_SCALE: usize = 10; // 图示时间缩放比例（每个字符代表 10ms）

fn get_time_u() -> usize {
    get_time() as usize
}//时间获取辅助

// 模拟各哲学家每轮思考和进餐持续时间（ms）
const ARR: [[usize; ROUND * 2]; N] = [
    [70, 80, 100, 40, 50, 60, 20, 40],
    [30, 60, 20, 70, 100, 10, 30, 60],
    [50, 20, 90, 20, 40, 60, 120, 40],
    [50, 100, 60, 50, 80, 60, 20, 90],
    [60, 10, 60, 60, 20, 50, 60, 20],
];
```

前期准备

```rust
fn philosopher_dining_problem(id: *const usize) {// 哲学家行为函数
    let id = unsafe { *id };
    let left = id;
    let right = if id == N - 1 { 0 } else { id + 1 };// // 为避免死锁，总是先拿编号小的锁再拿大的
    let min = if left < right { left } else { right };
    let max = left + right - min;
    for round in 0..ROUND {
        // --- 思考阶段 ---
        unsafe {
            THINK[id][2 * round] = get_time_u();
        }
        sleep_blocking(ARR[id][2 * round]);
        unsafe {
            THINK[id][2 * round + 1] = get_time_u();
        }
        // --- 等待并占用两把叉子（互斥锁） ---
        mutex_lock(min);
        mutex_lock(max);
        // --- 进餐阶段 ---
        unsafe {
            EAT[id][2 * round] = get_time_u();
        }
        sleep_blocking(ARR[id][2 * round + 1]);
        unsafe {
            EAT[id][2 * round + 1] = get_time_u();
        }
        // 释放锁（叉子）
        mutex_unlock(max);
        mutex_unlock(min);
    }
    exit(0)
}
```

哲学家行为函数

```rust
#[no_mangle]
pub fn main() -> i32 {
    println!("Here comes {} philosophers!", N);
    let mut v = Vec::new();
    let ids: Vec<_> = (0..N).collect();
    let start = get_time_u();
    // 创建N 个互斥锁（代表N 把叉子）
    for i in 0..N {
        assert_eq!(mutex_blocking_create(), i as isize);
        v.push(thread_create(// 创建N 个互斥锁（代表N 把叉子）
            philosopher_dining_problem as usize,
            &ids.as_slice()[i] as *const _ as usize,
        ));
    }
     // 等待所有线程结束
    for tid in v.iter() {
        waittid(*tid as usize);
    }
    let time_cost = get_time_u() - start;
    println!("time cost = {}", time_cost);
    println!("'-' -> THINKING; 'x' -> EATING; ' ' -> WAITING ");
    for id in (0..N).into_iter().chain(0..=0) {
        print!("#{}:", id);
        for j in 0..time_cost / GRAPH_SCALE {
            let current_time = j * GRAPH_SCALE + start;
            if (0..ROUND).any(|round| unsafe { // 判断此刻是思考、进餐还是等待
                let start_thinking = THINK[id][2 * round];
                let end_thinking = THINK[id][2 * round + 1];
                start_thinking <= current_time && current_time <= end_thinking
            }) {
                print!("-");
            } else if (0..ROUND).any(|round| unsafe {
                let start_eating = EAT[id][2 * round];
                let end_eating = EAT[id][2 * round + 1];
                start_eating <= current_time && current_time <= end_eating
            }) {
                print!("x");
            } else {
                print!(" ");
            };
        }
        println!("");
```

Main 函数逻辑

对于测试程序，我们可以直接使用作者在文件夹中给予我们的测试程序，我们可以在里面添加一些输出以表明该测试程序的开始以及最后通过的通知，然后我们运行该测试程序，能得到结果：

```
>> ch8b_phil_din_mutex
Here comes 5 philosophers!
time cost = 722
'-' -> THINKING; 'x' -> EATING; ' ' -> WAITING
#0: -------                 xxxxxxxx----------       xxxx-----  xxxxxx--xxx
#1: ---xxxxxx--       xxxxxxx----------    x---xxxxxx
#2: -----          xx---------xx----xxxxxx-----------        xxxx
#3: -----xxxxxxxxxx------xxxxx--------    xxxxxx--   xxxxxxxxx
#4: ------         x------          xxxxxx--    xxxxx------   xx
#0: -------                 xxxxxxxx----------       xxxx-----  xxxxxx--xxx
philosopher dining problem with mutex test passed!
```

看到测试通过。

2 进一步扩展内核功能，在内核线程中支持同步互斥机制，实现内核线程用的 mutex, semaphore, cond-var。

答：

扩展内核，我们可以增加相应的系统调用来实现该扩展：

```
pub const SYSCALL_WAITTID: usize = 462;
pub const SYSCALL_MUTEX_CREATE: usize = 463;
pub const SYSCALL_MUTEX_LOCK: usize = 464;
pub const SYSCALL_MUTEX_UNLOCK: usize = 466;
pub const SYSCALL_SEMAPHORE_CREATE: usize = 467;
pub const SYSCALL_SEMAPHORE_UP: usize = 468;
pub const SYSCALL_ENABLE_DEADLOCK_DETECT: usize = 469;
pub const SYSCALL_SEMAPHORE_DOWN: usize = 470;
pub const SYSCALL_CONDVAR_CREATE: usize = 471;
pub const SYSCALL_CONDVAR_SIGNAL: usize = 472;
pub const SYSCALL_CONDVAR_WAIT: usize = 473;
```

我们新增加以上的系统调用号声明

```
        SYSCALL_MUTEX_CREATE => sys_mutex_create(args[0] == 1),
        SYSCALL_MUTEX_LOCK => sys_mutex_lock(args[0]),
        SYSCALL_MUTEX_UNLOCK => sys_mutex_unlock(args[0]),
        SYSCALL_SEMAPHORE_CREATE => sys_semaphore_create(args[0]),
        SYSCALL_SEMAPHORE_UP => sys_semaphore_up(args[0]),
        SYSCALL_ENABLE_DEADLOCK_DETECT => sys_enable_deadlock_detect(args[0]),
        SYSCALL_SEMAPHORE_DOWN => sys_semaphore_down(args[0]),
        SYSCALL_CONDVAR_CREATE => sys_condvar_create(),
        SYSCALL_CONDVAR_SIGNAL => sys_condvar_signal(args[0]),
        SYSCALL_CONDVAR_WAIT => sys_condvar_wait(args[0], args[1]),
```

```rust
pub fn sys_mutex_create(blocking: bool) -> isize {
    syscall(SYSCALL_MUTEX_CREATE, [blocking as usize, 0, 0])
}

pub fn sys_mutex_lock(id: usize) -> isize {
    syscall(SYSCALL_MUTEX_LOCK, [id, 0, 0])
}

pub fn sys_mutex_unlock(id: usize) -> isize {
    syscall(SYSCALL_MUTEX_UNLOCK, [id, 0, 0])
}

pub fn sys_semaphore_create(res_count: usize) -> isize {
    syscall(SYSCALL_SEMAPHORE_CREATE, [res_count, 0, 0])
}

pub fn sys_semaphore_up(sem_id: usize) -> isize {
    syscall(SYSCALL_SEMAPHORE_UP, [sem_id, 0, 0])
}

pub fn sys_enable_deadlock_detect(enabled: usize) -> isize {
    syscall(SYSCALL_ENABLE_DEADLOCK_DETECT, [enabled, 0, 0])
}

pub fn sys_semaphore_down(sem_id: usize) -> isize {
    syscall(SYSCALL_SEMAPHORE_DOWN, [sem_id, 0, 0])
}

pub fn sys_condvar_create(_arg: usize) -> isize {
    syscall(SYSCALL_CONDVAR_CREATE, [_arg, 0, 0])
}

pub fn sys_condvar_signal(condvar_id: usize) -> isize {
    syscall(SYSCALL_CONDVAR_SIGNAL, [condvar_id, 0, 0])
}

pub fn sys_condvar_wait(condvar_id: usize, mutex_id: usize) -> isize {
    syscall(SYSCALL_CONDVAR_WAIT, [condvar_id, mutex_id, 0])
}
```

系统调用函数参数类型如上。

```rust
pub fn sys_mutex_lock(mutex_id: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_mutex_lock",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );// 打日志 (trace 级别)，记录当前进程的 PID 和当前线程的 TID，以便调试追踪。
    let process = current_process();
    let process_inner = process.inner_exclusive_access();
    //获取对当前进程结构的独占访问 (相当于一把互斥锁)，以安全地读取其内部状态。
    let mutex = Arc::clone(process_inner.mutex_list[mutex_id].as_ref().unwrap());//从进程的 `mutex_list` 中取出指定 ID 的 Mutex 对象
    drop(process_inner);
    drop(process);//主动释放对进程结构的独占访问锁
    mutex.lock();//调用该 `Arc<Mutex<...>>` 的 `lock()` 方法。
    0
}
```

mutex_lock 系统调用的具体实现

mutex_lock，该函数的思路很简单：先从当前进程安全地取出并克隆出要加锁的互斥体句柄，释放对进程结构的锁后，再调用该句柄的 lock() 阻塞等待加锁，最后返回成功。

```rust
pub fn sys_mutex_unlock(mutex_id: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_mutex_unlock",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );
    let process = current_process();
    let process_inner = process.inner_exclusive_access();
    let mutex = Arc::clone(process_inner.mutex_list[mutex_id].as_ref().unwrap());
    drop(process_inner);
    drop(process);
    mutex.unlock();
    0
}
```

mutex_unlock，可以看到和上面的 mutex_lock 的实现基本差不多，只是把上锁改成了解锁。

```rust
pub fn sys_semaphore_create(res_count: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_semaphore_create",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );//使用 trace! 宏在内核日志中记录当前进程的 PID、线程的 TID，以及此次调用的名称 sys_semaphore_create，便于后续调试和审计。
    let process = current_process();
    let mut process_inner = process.inner_exclusive_access();//获取并锁定进程内部结构
    let id = if let Some(id) = process_inner
        .semaphore_list
        .iter()
        .enumerate()
        .find(|(_, item)| item.is_none())
        .map(|(id, _)| id)
    {
        process_inner.semaphore_list[id] = Some(Arc::new(Semaphore::new(res_count)));
        id
    } else {
        process_inner
            .semaphore_list
            .push(Some(Arc::new(Semaphore::new(res_count))));
        process_inner.semaphore_list.len() - 1
    };//在 semaphore_list 中分配一个新信号量槽
    id as isize//返回信号量 ID
}
```

sys_semaphore_create 的实现

该函数的实现思路如下：

首先，对于日志与上下文准备，在一切操作之前，我们先用 trace! 宏记录当前进程的 PID 和线程 TID，以及调用名称 sys_semaphore_create，方便后续定位问题。然后通过 current_process() 和 inner_exclusive_access() 安全地拿到对当前进程内部状态的独占访问，确保在修改信号量列表期间不会被其他核内并发操作干扰。接着，寻找或分配槽位进程维护了一个 Vec<Option<Arc<Semaphore>>>——semaphore_list。这里的 Option 表示某个槽位要么存放一个信号量句柄（Some(Arc<Semaphore>)），要么空置（None）。我们先用 iter().enumerate().find(|(_, item)| item.is_none()) 快速查找第一个空槽：如果找到了，就在这个位置创建一个新的 Semaphore（初始资源计数为 res_count），包装在 Arc 里，然后写回该槽。如果整个列表都没有空位，则直接在末尾 push 一个新的 Some(Arc::new(Semaphore::new(res_count)))，完成扩容。引用管理与返回

之所以用 Arc<Semaphore>，是为了让内核内多个引用可以安全地共享同一个信号量对象，并自动管理其生命周期。函数最后把该信号量在列表中的索引（即槽位号）转成 isize 返回给用户态，后续对该信号量的 wait、post 或 destroy 都可以通过这个索引来定位。

```rust
pub fn sys_semaphore_up(sem_id: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_semaphore_up",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );//日志追踪
    let process = current_process();
    let process_inner = process.inner_exclusive_access();//安全访问进程结构
    let sem = Arc::clone(process_inner.semaphore_list[sem_id].as_ref().unwrap());//克隆信号量句柄
    drop(process_inner);//释放对进程内部数据的锁
    sem.up();
    0
}
```

semaphore_up 的实现

该函数记录调用者信息后，从当前进程的信号量列表中克隆出对应的 Semaphore，释放进程内部锁，再调用它的 up() 方法增加资源计数，最后返回成功。

```rust
pub fn sys_semaphore_down(sem_id: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_semaphore_down",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );
    let process = current_process();
    let process_inner = process.inner_exclusive_access();//获取进程上下文并加锁
    let sem = Arc::clone(process_inner.semaphore_list[sem_id].as_ref().unwrap());
    drop(process_inner);//释放对进程内部结构的锁
    sem.down();
    0
}
```

semaphore_down 的实现

down 和 up 的实现基本一致。只是把 V 操作（释放 / 归还）改成了 P（等待 / 申请）。

类似的，我们还需要实现 cond-var 的三个函数，condvar_create()，condvar_signal，condvar_wait。

```rust
pub fn sys_condvar_create() -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_condvar_create",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );
    let process = current_process();
    let mut process_inner = process.inner_exclusive_access();
    let id = if let Some(id) = process_inner
        .condvar_list
        .iter()
        .enumerate()
        .find(|(_, item)| item.is_none())
        .map(|(id, _)| id)
    {
        process_inner.condvar_list[id] = Some(Arc::new(Condvar::new()));
        id
    } else {
        process_inner
            .condvar_list
            .push(Some(Arc::new(Condvar::new())));
        process_inner.condvar_list.len() - 1
    };
    id as isize
}
```

condvar_create()

```rust
pub fn sys_condvar_signal(condvar_id: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_condvar_signal",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );
    let process = current_process();
    let process_inner = process.inner_exclusive_access();
    let condvar = Arc::clone(process_inner.condvar_list[condvar_id].as_ref().unwrap());
    drop(process_inner);
    condvar.signal();
    0
}
```

condvar_signal

```rust
pub fn sys_condvar_wait(condvar_id: usize, mutex_id: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_condvar_wait",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );
    let process = current_process();
    let process_inner = process.inner_exclusive_access();
    let condvar = Arc::clone(process_inner.condvar_list[condvar_id].as_ref().unwrap());
    let mutex = Arc::clone(process_inner.mutex_list[mutex_id].as_ref().unwrap());
    drop(process_inner);
    condvar.wait(mutex);
    0
}
```

condvar_wait

之后，我们编写测试程序：

```rust
unsafe fn first() -> ! {
    sleep(10);
    println!("First work, Change A --> 1 and wakeup Second");
    mutex_lock(MUTEX_ID);
    A = 1;
    condvar_signal(CONDVAR_ID);
    mutex_unlock(MUTEX_ID);
    exit(0)
}
```

第一个 condvar

```rust
unsafe fn second() -> ! {
    println!("Second want to continue,but need to wait A=1");
    mutex_lock(MUTEX_ID);
    while A == 0 {
        println!("Second: A is {}", A);
        condvar_wait(CONDVAR_ID, MUTEX_ID);
    }
    mutex_unlock(MUTEX_ID);
    println!("A is {}, Second can work now", A);
    exit(0)
}
```

第二个 condvar，我们使用 wait 来使其等待第一个作业的信号。

```rust
pub fn main() -> i32 {
    // create condvar & mutex
    assert_eq!(condvar_create() as usize, CONDVAR_ID);
    assert_eq!(mutex_blocking_create() as usize, MUTEX_ID);
    // create threads
    let threads = vec![
        thread_create(first as usize, 0),
        thread_create(second as usize, 0),
    ];
    // wait for all threads to complete
    for thread in threads.iter() {
        waittid(*thread as usize);
    }
    println!("test_condvar passed!");
    0
}
```

主函数，进行线程的创建以及测试函数的执行，如果一切顺利，最后打印出测试通过。

运行该测试程序，得到结果：

```
>> ch8b_test_condvar
Second want to continue,but need to wait A=1
Second: A is 0
First work, Change A --> 1 and wakeup Second
A is 1, Second can work now
test_condvar passed!
```

看到测试通过。

3. *** 扩展内核功能，在内核中支持内核线程。

答：和上面的扩展内核功能相似，我们同样的使用增加系统调用的方法来完成这个任务。

```rust
        // SYSCALL_TASK_INFO => sys_task_info(args[0] as *mut TaskInfo),
        // SYSCALL_MMAP => sys_mmap(args[0], args[1], args[2]),
        // SYSCALL_MUNMAP => sys_munmap(args[0], args[1]),
        // SYSCALL_SBRK => sys_sbrk(args[0] as i32),
        // SYSCALL_SPAWN => sys_spawn(args[0] as *const u8),
        // SYSCALL_SET_PRIORITY => sys_set_priority(args[0] as isize),
        SYSCALL_THREAD_CREATE => sys_thread_create(args[0], args[1]),
```

```rust
pub const SYSCALL_PIPE: usize = 59;
pub const SYSCALL_DUP: usize = 24;
pub const SYSCALL_THREAD_CREATE: usize = 460;
```

创建线程的系统调用声明

我们在 syscall 文件夹下专门创建一个新的 rs 文件，thread.rs 文件来实现该系统调用。

```rust
pub fn sys_thread_create(entry: usize, arg: usize) -> isize {
    trace!(
        "kernel:pid[{}] tid[{}] sys_thread_create",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );//日志追踪
    let task = current_task().unwrap();
    let process = task.process.upgrade().unwrap();
    // create a new thread
    let new_task = Arc::new(TaskControlBlock::new(
        Arc::clone(&process),
        task.inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .ustack_base,
        true,
    ));//构造新线程的 TCB
    add_task(Arc::clone(&new_task));//将新线程加入调度器
    let new_task_inner = new_task.inner_exclusive_access();
    let new_task_res = new_task_inner.res.as_ref().unwrap();
    let new_task_tid = new_task_res.tid;//读取新线程的基本信息
    let mut process_inner = process.inner_exclusive_access();
    // 把新线程加入进程的线程列表
    let tasks = &mut process_inner.tasks;
    while tasks.len() < new_task_tid + 1 {
        tasks.push(None);
    }
    tasks[new_task_tid] = Some(Arc::clone(&new_task));
    let new_task_trap_cx = new_task_inner.get_trap_cx();//设置用户态初始 TrapContext
    *new_task_trap_cx = TrapContext::app_init_context(
        entry,
        new_task_res.ustack_top(),
        kernel_token(),
        new_task.kstack.get_top(),
        trap_handler as usize,
    );
    (*new_task_trap_cx).x[10] = arg;
    new_task_tid as isize// 返回新线程 TID
}
```

创建线程的函数实现

该函数在内核中为当前进程创建一个新的线程(Task),将其加入调度器并进程的线程列表，设置好用户态的初始 TrapContext（即寄存器上下文），最后返回新线程的 TID。日志 → 获取上下文：打印日志、拿到当前任务和进程。首先做新线程构造，即用父进程和用户栈地址创建一个新的 TaskControlBlock，并加到调度器。接着线程列表登记：将新线程插入所在进程的 tasks 列表。再之后是 TrapContext 初始化，为新线程构造一个符合用户程序启动的寄存器上下文，并设置函数参数。最后我们进行了返回 TID，将新线程的标识符返回给用户态。这样就完成了一个从内核创建、调度到用户态可运行的完整线程启动流程。

之后，我们还要实现一个 sys_waittid 来进行线程等待，来完善线程的实现：

```rust
pub fn sys_waittid(tid: usize) -> i32 {
    trace!(
        "kernel:pid[{}] tid[{}] sys_waittid",
        current_task().unwrap().process.upgrade().unwrap().getpid(),
        current_task()
            .unwrap()
            .inner_exclusive_access()
            .res
            .as_ref()
            .unwrap()
            .tid
    );
    let task = current_task().unwrap();
    let process = task.process.upgrade().unwrap();
    let task_inner = task.inner_exclusive_access();
    let mut process_inner = process.inner_exclusive_access();//获取当前任务与进程，并加锁
    if task_inner.res.as_ref().unwrap().tid == tid {
        return -1;
    }//禁止自我等待
    let mut exit_code: Option<i32> = None;
    let waited_task = process_inner.tasks[tid].as_ref();
    if let Some(waited_task) = waited_task {
        if let Some(waited_exit_code) = waited_task.inner_exclusive_access().exit_code {
            exit_code = Some(waited_exit_code);
        }
    } else {
        return -1;
    }//检查目标线程存在性及退出状态
    if let Some(exit_code) = exit_code {
        // 已退出，收回槽位
        process_inner.tasks[tid] = None;
        exit_code
    } else {
        // 等待的线程不存在
        -2
```

线程等待函数的实现

问答题：

1 什么是并行？什么是并发
答：并发是指系统能够同时处理多个任务，任务之间交替执行；并行是指多个任务在多个处理器上真正同时运行。并发关注任务的切换和协调，并行关注任务的同时执行，二者可以同时存在。
2 ** Linux 的多线程应用程序使用的锁（例如 pthread_mutex_t）不是自旋锁，当上锁失败时会切换到其它进程执行。分析它和自旋锁的优劣，并说明为什么它不用自旋锁？
答：在 Linux 的多线程应用程序中，常用的 pthread_mutex_t 是一种阻塞锁，它在竞争失败时会主动让出 CPU 并进入休眠状态，等待锁可用后再被唤醒。而自旋锁（spinlock）在竞争锁失败时会持续占用 CPU 轮询锁状态，直到获取成功。

从性能和适用场景来看，这两种锁各有优劣。自旋锁的优点是没有上下文切换的开销，适合临界区非常短、线程竞争激烈但等待时间极短的场景（如内核中断上下文）；缺点是如果锁被持有时间较长，会浪费 CPU 资源，影响系统吞吐量。相反，阻塞锁的优势在于节省 CPU，适合临界区执行时间不确定或较长的情况，缺点是涉及系统调用和调度器参与，可能带来上下文切换的额外开销。

Linux 用户态线程库选择使用阻塞锁（pthread_mutex_t）而不是自旋锁，主要是出于通用性、效率和系统调度协同的考虑。大多数用户态程序的临界区不会极短，盲目使用自旋锁不仅无益，还可能严重浪费 CPU。阻塞锁还支持更高级的调度控制（如优先级继承），能够有

效避免优先级反转等复杂问题，更适合构建健壮和可扩展的多线程应用。
3 *** 程序在运行时具有两种性质：safety: something bad will never happen；liveness: something good will eventually occur. 分析并证明 Peterson 算法的 safety 和 liveness 性质。
答：Peterson 算法在两进程互斥中同时满足 safety 和 liveness 两个性质：它通过两个标志变量和一个 turn 变量保证在任何时刻最多只有一个进程能进入临界区，从而满足 safety（不会发生两个进程同时进入临界区的“坏事”）；同时，算法设计确保若某个进程持续请求进入临界区，最终一定能进入，即满足 liveness（“好事”最终会发生），因此该算法既安全又无饥饿，适合作为互斥控制的基础模型。
4 ** 条件变量的 Wait 操作为什么必须关联一个锁？
答：条件变量的 Wait 操作必须关联一个锁，是因为等待线程在进入等待状态前，需要先释放锁以允许其他线程修改条件状态；而当等待结束被唤醒时，线程又必须重新获取该锁，确保在检查和修改共享数据时的互斥性和数据一致性。这种“释放锁-等待-重新获取锁”的机制防止了竞态条件，保证了线程间的正确同步和安全访问。
5 * 死锁的必要条件是什么？
答：死锁的四个必要条件是：
互斥条件：至少有一个资源是非共享的，某个进程占用该资源时，其他进程只能等待。
占有且等待：一个进程至少占有一个资源，同时等待获取其他被别的进程占有的资源。
不剥夺条件：资源不能被强制从进程中剥夺，只能由占有它的进程主动释放。
环路等待：存在一组进程，形成环状链，每个进程都在等待下一个进程持有的资源。
6 * 什么是死锁预防，举例并分析
答：死锁预防是通过破坏死锁发生的四个必要条件之一或多个，来避免系统进入死锁状态的一种策略。它在资源分配时提前采取措施，确保死锁条件永远不会同时成立。
举例与分析：
破坏互斥条件：将可共享资源设计为多进程共享，比如读者-写者锁允许多个读者同时访问，减少互斥资源。
破坏占有且等待条件：要求进程一次性申请所需所有资源，若不能全部获得则不占有任何资源，防止“占有且等待”情况。例如，银行家算法中，进程必须声明最大需求，系统根据安全状态分配资源。
破坏不剥夺条件：允许系统强制回收资源，如当进程等待资源时，系统可以抢占其他进程已占有的资源并重新分配，避免资源长期被占用。
破坏环路等待条件：为所有资源分配一个固定的顺序，进程必须按序申请资源，避免循环等待的发生。
7 ** 描述银行家算法如何判断安全性。
答：银行家算法判断系统安全性的核心是模拟资源分配过程，检查是否存在一个进程执行顺序，使所有进程都能顺利获得所需资源并完成，最终释放资源。具体步骤有，初始化：记录当前系统的可用资源数（Work），并标记所有进程为未完成（Finish[i] = false）。寻找进程：在未完成的进程中，找到一个其剩余需求（Need[i]）不超过当前可用资源（Work）的进程 i。模拟执行：假设进程 i 执行完成，释放其占有资源，将这些资源加回 Work，并标记 Finish[i] = true。重复步骤 2 和 3，直到所有进程都被标记为完成，或者找不到满足条件的进程。判断：如果所有进程 Finish[i]均为 true，系统处于安全状态；否则系统不安全，分配资源可能导致死锁。
8 信号量结构中的整数分别为+n、0、-n 的时候，各自代表什么状态或含义？
答：+n 表示还有 n 个可用资源，0 表示所有可用资源恰好耗尽，-n 则表示有 n 个进程申请

了资源但无资源可用，被阻塞。

实践题：

背景：在智能体大赛平台 Saiblo 网站上每打完一场双人天梯比赛后需要用 ELO 算法更新双方比分。由于 Saiblo 的评测机并发性很高，且 ELO 算法中的分值变动与双方变动前的分数有关，因此更新比分前时必须先为两位选手加锁。
作业：请模拟一下上述分数更新过程，简便起见我们简化为有 p 位选手参赛（编号 [0, p) 或 [1, p] ），初始分值为 1000 分，有 m 个评测机线程（生产者）给出随机的评测结果（两位不同选手的编号以及胜负结果，结果可能为平局），有 n 个 worker 线程（消费者）获取结果队列并更新数据库（全局变量等共享数据）记录的分数。m 个评测机各自模拟 k 场对局结果后结束线程，全部对局比分更新完成后主线程打印每位选手最终成绩以及所有选手分数之和。
上述参数 p、m、n、k 均为可配置参数（命令行传参或程序启动时从 stdin 输入）。
简便起见不使用 ELO 算法，简化更新规则为：若不为平局，当 胜者分数 >= 败者分数 时胜者 +20，败者 -20，否则胜者 +30，败者 -30；若为平局，分高者 -10，分低者+10（若本就同分保持则不变）。
根据题目提示，我们要实现 eventfd 的系统调用并且以此来实现该题目算法。

- syscall ID: 290
- 功能：创建一个 eventfd， eventfd 标准接口 。
- C 接口： `int eventfd(unsigned int initval, int flags)`
- Rust 接口： `fn eventfd(initval: u32, flags: i32) -> i32`
- **参数：**
  - initval: 计数器的初值。
  - **flags: 可以设置为 0 或以下两个 flag 的任意组合（按位或）：**
    - EFD_SEMAPHORE (1)： 设置该 flag 时，将以信号量模式创建 eventfd 。
    - EFD_NONBLOCK (2048)： 若设置该 flag ，对 eventfd 读写失败时会返回 -2 ，否则将阻塞等待直至读或写操作可执行为止。
- **说明：**
  - 通过 *write* 写入 eventfd 时，缓冲区大小必须为 8 字节。
  - 进程 *fork* 时，子进程会继承父进程创建的 eventfd ，且指向同一个计数器。
- 返回值：如果出现了错误则返回 -1，否则返回创建成功的 eventfd 编号。
- **可能的错误**
  - flag 不合法。
  - 创建的文件描述符数量超过进程限制

以此为根据我们来创建系统调用。

```
pub const SYSCALL_CONDVAR_SIGNAL: usize = 472;
pub const SYSCALL_CONDVAR_WAIT: usize = 473;
pub const SYSCALL_EVENTFD2: usize = 290;
```

系统调用号

```
SYSCALL_CONDVAR_WAIT => sys_condvar_wait(args[0], args[1]),
SYSCALL_EVENTFD2 => sys_eventfd2(args[0] as u32, args[1] as i32),
```

系统调用函数

```rust
/// eventfd2 syscall: 返回新 fd 或 -1
pub fn sys_eventfd2(initval: u32, flags: i32) -> isize {
    if let Some(evtfd) = EventFd::new(initval, flags) {
        // 把 kernel 对象当 File trait 对象包装
        let file_ref: Arc<dyn File> = Arc::new(evtfd);

        let process = current_process();
        let mut inner = process.inner_exclusive_access();
        let fd = inner
            .fd_table
            .iter()
            .position(Option::is_none)
            .expect("fd table full");
        inner.fd_table[fd] = Some(file_ref);
        fd as isize
    } else {
        -1
    }
}
```

系统调用的功能实现。

完成创建之后，我们来实现题目要求的算法。在我们的操作系统文件管理系统文件夹下我们创建文件 eventfd.rs 文件，在里面实现算法。

```rust
use core::sync::atomic::{AtomicU64, Ordering};
use alloc::sync::Arc;
use crate::config::{EFD_NONBLOCK, EFD_SEMAPHORE};
use crate::sync::{Mutex, MutexBlocking, MutexSpin, Condvar};
use crate::fs::{File, UserBuffer, Stat, StatMode};
```

要用到的外部调用

```rust
pub struct EventFd {
    counter: AtomicU64,
    flags: i32,
    /// 用于阻塞唤醒的互斥锁
    mutex: Arc<dyn Mutex>,
    condvar: Condvar,
}
```

定义 eventfd 对象

```rust
impl EventFd {
    /// 新建一个 eventfd
    pub fn new(initval: u32, flags: i32) -> Option<Self> {
        // 仅允许这两个 flag
        if flags & !(EFD_NONBLOCK | EFD_SEMAPHORE) != 0 {
            return None;
        }
        // 选用阻塞或自旋锁
        let m: Arc<dyn Mutex> = if flags & EFD_SEMAPHORE != 0 {
            Arc::new(MutexSpin::new())
        } else {
            Arc::new(MutexBlocking::new())
        };
        Some(EventFd {
            counter: AtomicU64::new(initval as u64),
            flags,
            mutex: m,
            condvar: Condvar::new(),
        })
    }
}
```

Eventfd 对象的构造函数

```rust
fn read(&self, buf: UserBuffer) -> usize {
    // 如果是非阻塞模式且计数器为 0，立即返回 0
    if (self.flags & EFD_NONBLOCK) != 0 && self.counter.load(Ordering::SeqCst) == 0 {
        return 0;
    }
    // 阻塞模式：在 counter>0 之前循环等待
    if self.counter.load(Ordering::SeqCst) == 0 {
        // 上锁
        self.mutex.lock();
        while self.counter.load(Ordering::SeqCst) == 0 {
            // 传入同一个 Arc<dyn Mutex> 克隆，用于 condvar.wait
            self.condvar.wait(self.mutex.clone());
        }
        // 解锁：如果你的 Condvar.wait 自动解锁/重锁则可省，此处根据实现决定
        self.mutex.unlock();
    }

    // 取值
    let val = if (self.flags & EFD_SEMAPHORE) != 0 {
        // 信号量模式：每次减 1 并返回 1
        self.counter.fetch_sub(1, Ordering::SeqCst);
        1u64
    } else {
        // 普通模式：返回当前值并清零
        self.counter.swap(0, Ordering::SeqCst)
    };

    // 写回用户缓冲
    let bytes = val.to_ne_bytes();
    let mut written = 0;
    let mut iter = buf.into_iter();
    for &b in &bytes {
        if let Some(byte_ref) = iter.next() {
            unsafe { *byte_ref = b; }
            written += 1;
        } else {
            break;
        }
    }
    written
}
```

Read 部分的逻辑

```rust
    fn write(&self, buf: UserBuffer) -> usize {
        // 从用户缓冲读取 8 字节
        let mut bytes = [0u8; 8];
        let mut read = 0;
        let mut iter = buf.into_iter();
        while read < 8 {
            if let Some(byte_ref) = iter.next() {
                bytes[read] = unsafe { *byte_ref };
                read += 1;
            } else {
                break;
            }
        }
        if read < 8 {
            return 0;
        }

        // 更新计数器
        let v = u64::from_ne_bytes(bytes);
        self.counter.fetch_add(v, Ordering::SeqCst);
        // 唤醒所有等待者
        self.condvar.signal();

        8
    }
}
```

Write 部分的逻辑

```rust
impl EventFd {
    pub fn stat(&self) -> Stat {
        Stat {
            dev: 0,
            ino: 0,
            mode: StatMode::FILE,
            nlink: 1,
            pad: [0; 7],
        }
    }
}
```

为 EventFd 提供一个 stat() 方法

完成题目算法的实现之后，我们在 user 里面创建 eventfd_test 的测试文件，来进行题目中要求的测试内容。

```rust
#![no_std]
#![no_main]
#![feature(custom_test_frameworks)]
#![reexport_test_harness_main = "test_main"]
#![test_runner(test_runner)]

extern crate alloc;
use alloc::vec::Vec;
use core::arch::asm;
use user_lib::{exit, println, print};
use user_lib::{semaphore_create, semaphore_down, semaphore_up};
use user_lib::{thread_create, waittid};
```

外部引用

```rust
const BUFFER_SIZE: usize = 8;
const PRODUCER_COUNT: usize = 100;
const NUMBER_PER_PRODUCER: usize = 100;
```

```rust
const P: usize = 100;
const PRODUCER_THREADS: usize = PRODUCER_COUNT;
const CONSUMER_THREADS: usize = 1;
```

设置参数 m，n，p

```rust
const SEM_MUTEX: usize = 0;
const SEM_EMPTY: usize = 1;
const SEM_EXISTED: usize = 2;
```

设置信号量编号

```rust
fn pack(u: usize, v: usize, outcome: u8) -> u64 {
    ((u as u64) << 32) | ((v as u64) << 8) | outcome as u64
}
fn unpack(data: u64) -> (usize, usize, u8) {
    let u = (data >> 32) as usize;
    let v = ((data >> 8) & 0x00FF_FFFF) as usize;
    let o = (data & 0xFF) as u8;
    (u, v, o)
}
```

打包和拆包的逻辑

```rust
unsafe fn producer(id_ptr: *const usize) -> ! {
    let id = *id_ptr;
    for _ in 0..NUMBER_PER_PRODUCER {
        // 随机选手与结果
        let u = id;
        let v = (id + 1) % P;
        let outcome = (id as u8) % 3;
        let m = pack(u, v, outcome);
        semaphore_down(SEM_EMPTY);
        semaphore_down(SEM_MUTEX);
        BUFFER[FRONT] = m;
        FRONT = (FRONT + 1) % BUFFER_SIZE;
        semaphore_up(SEM_MUTEX);
        semaphore_up(SEM_EXISTED);
    }
    exit(0)
}
```

生产者逻辑，也就是比赛的裁判，进行比赛双方的选手选择以及结果判定。

```rust
unsafe fn consumer(_arg: *const usize) -> ! {
    for _ in 0..(PRODUCER_COUNT * NUMBER_PER_PRODUCER) {
        semaphore_down(SEM_EXISTED);
        semaphore_down(SEM_MUTEX);
        let m = BUFFER[TAIL];
        TAIL = (TAIL + 1) % BUFFER_SIZE;
        semaphore_up(SEM_MUTEX);
        semaphore_up(SEM_EMPTY);

        let (u, v, outcome) = unpack(m);
        // 更新分数
        let su = &mut SCORES[u];
        let sv = &mut SCORES[v];
        match outcome {
            0 => {
                if *su > *sv { *su -= 10; *sv += 10; }
                else if *su < *sv { *su += 10; *sv -= 10; }
            }
            1 => {
                if *su >= *sv { *su += 20; *sv -= 20; }
                else          { *su += 30; *sv -= 30; }
            }
            2 => {
                if *sv >= *su { *sv += 20; *su -= 20; }
                else          { *sv += 30; *su -= 30; }
            }
            _ => {}
        }
    }
    // 打印结果
    for i in 0..P {
        print!("{} ", SCORES[i]);
    }
    println!("");
    exit(0)
}
```

消费者逻辑，从缓冲区读取比赛结果并且更新选手的分数。

```rust
pub fn main() -> i32 {
    // 创建信号量
    assert_eq!(semaphore_create(1) as usize, SEM_MUTEX);
    assert_eq!(semaphore_create(BUFFER_SIZE) as usize, SEM_EMPTY);
    assert_eq!(semaphore_create(0) as usize, SEM_EXISTED);

    // 创建生产者线程
    let ids: Vec<usize> = (0..PRODUCER_COUNT).collect();
    let mut tids: Vec<usize> = Vec::new();
    for i in 0..PRODUCER_COUNT {
        // 把 isize 强制转成 usize
        let t = thread_create(producer as usize, &ids[i] as *const _ as usize) as usize;
        tids.push(t);
    }

    // 创建消费者线程，也要转成 usize
    let t_consumer = thread_create(consumer as usize, 0) as usize;
    tids.push(t_consumer);

    // 等待所有线程
    for &t in &tids {
        waittid(t);
    }

    println!("test_eventfd passed!");
    0
}
```

主函数通道

运行该测试程序，我们可以看到结果：

```
>> ch8_eventfd_test
2320 2370 -3000 2990 3010 -3000 2990 3010 -3000 2990 3010 -3000 2990 3010 -3000
3000 3000 -3000 2990 3010 -3000 3000 3000 -3000 3000 3000 -3000 3000 3000 -3000
2990 3010 -3000 3000 3000 -3000 3000 3000 -3000 3000 3000 -3000 2990 3010 -3000
2990 3010 -3000 2990 3010 -3000 2990 3010 -3000 2990 3010 -3000 2990 3010 -3000
2990 3010 -3000 3000 3000 -3000 2990 3010 -3000 2990 3010 -3000 2990 3010 -3000
3000 3000 -3000 2990 3010 -3000 3000 3000 -3000 3000 3000 -3000 3000 3000 -3000
3000 3000 -3000 2990 3010 -3000 3000 3000 -3000 2310
test_eventfd passed!
```

结果中打印出了总共 100 位选手最后各自的分数。