

北京邮电大学

逻辑/物理地址转换实验报告



课程名称: 操作系统
实验名称: 逻辑/物理地址转换
学 院: 计算机学院
班 级: 2022211312
学 号: 2022211404
姓 名: 唐梓楠

2024 年 12 月 23 日

目录

1 任务分析	2	4.4 重启并更换内核	14
1.1 进程	2	4.5 验证内核配置	15
1.2 临界区 (Critical Section)	2	5 系统框架及关键代码	16
1.3 Peterson 算法	3	5.1 main() 函数	16
1.4 逻辑地址与物理地址	4	5.1.1 分配共享内存	16
2 实验环境	5	5.1.2 初始化共享变量	18
3 设计思路	6	5.1.3 创建子进程	18
3.1 共享内存的创建和初始化	6	5.2 Peterson 算法: peterson_algorithm() 函数	19
3.2 Peterson 算法的实现	6	5.3 获取物理地址: virtual_to_physical_address() 函数	20
3.3 获取虚拟地址和物理地址	6	5.4 验证物理函数正确: verify_physical_address() 函数	21
3.4 虚拟地址结构	6	6 模块流程图	21
3.5 物理地址验证	9	7 测试	22
3.6 进程间同步和协作	9	A 程序文档 by doxygen	23
4 实现方法: 修改 Linux 内核	10		
4.1 获取内核源代码	10		
4.2 配置内核	11		
4.3 编译和安装自定义内核	13		

逻辑/物理地址转换实验报告

唐梓楠

2024 年 12 月 23 日

1 任务分析

由于 MacOS 以及 Windows 对于物理内存直接访问有着严格的限制，本实验在 Ubuntu 主机上完成，处于安全原因，由于 Linux 对物理地址直接访问文件 `/dev/mem` 的限制默认开启，只能查看前 1MB 的数据，见 [Linux Kernel Driver DataBase](#)，因此还需要修改内核实现。

1.1 进程

进程是操作系统中最基本的执行单元，是程序在其数据集上的一次执行过程。进程不仅包含程序代码，还包括程序计数器、寄存器、堆栈等信息。在现代操作系统中，进程是多任务的基础，通过调度系统可以让多个进程并行运行。

在本实验中，需要创建两个进程，且它们需要竞争进入临界区（Critical Section）。临界区是指多个进程共享的资源在同一时刻只能被一个进程访问的区域。

1.2 临界区（Critical Section）

临界区（Critical Section）是指一段程序代码，其中访问共享资源（如变量、文件、硬件等）时，多个进程或线程可能同时进入该区域，导致并发问题的发生。为了保证程序的正确性和数据的一致性，必须确保在同一时刻，只有一个进程或线程能够执行这段代码。否则，多个进程或线程可能会在没有适当同步的情况下对共享资源进行修改，从而引发竞态条件（race condition）和不可预期的结果。

在多进程或多线程环境下，操作系统需要采取措施来控制对临界区的访问，以防止出现这种竞态条件。通常，这些措施包括互斥锁、信号量、条件变量等同步机制。为了正确实现临界区的管理，

操作系统需要提供一个机制来确保只有一个进程能够进入临界区，其他进程则必须等待，直到当前进程完成对共享资源的访问并退出临界区。

临界区问题的核心在于**互斥性**。互斥性意味着在任何时刻，只有一个进程或线程能够访问临界区。没有互斥，就可能导致共享资源的状态不一致，比如两个进程同时修改同一个变量，可能导致数据丢失或错误的计算结果。因此，在多任务操作系统中，解决临界区问题是保证程序正确执行的关键。

要解决临界区问题，需要引入进程或线程间的同步机制，这样在任意时刻，只有一个进程能够进入临界区，其他进程则需要等待。在早期的操作系统设计中，经典的 **Peterson 算法**就是一种为了解决两个进程的临界区竞争问题而提出的算法，它通过使用共享变量来控制两个进程的访问顺序，确保互斥。

临界区问题的解决不仅涉及互斥性，还可能涉及其他性质，例如死锁避免、进程的公平性（防止饥饿），以及进程或线程的并发执行效率。在实际应用中，操作系统使用多种方法来确保这些性质得到满足，确保系统的稳定和可靠运行。

1.3 Peterson 算法

Peterson 算法是一种经典的用于解决两个进程之间临界区竞争问题的算法，它通过使用两个共享变量来控制进程之间的进入和退出临界区的顺序，从而确保互斥性。这个算法由 Gary Peterson 在 1981 年提出，旨在为两个进程提供一个简单且有效的方式来解决竞争条件的问题，保证在任何时刻只有一个进程可以进入临界区，从而避免出现数据竞争。

Peterson 算法的核心思想是利用两个共享变量——`flag[2]` 和 `turn`，来协调两个进程的执行顺序。`flag[i]` 用来表示进程 `i` 是否准备进入临界区，`turn` 则用来指示哪个进程应该进入临界区。具体地，`flag[i]` 是一个布尔变量，当进程 `i` 想要进入临界区时，它将 `flag[i]` 设置为 `true`，表示它有意愿进入临界区。而 `turn` 是一个标记，表示当前应该由哪个进程进入临界区，进程 `i` 会将 `turn` 设置为另一个进程的编号，从而让对方优先进入临界区。

当进程准备进入临界区时，首先它会将自己的 `flag[i]` 设置为 `true`，表示它想要进入临界区。然后，进程会将 `turn` 设置为另一个进程的编号，意图告诉对方如果它想要进入临界区，它应该先进入。接下来，进程会检查另一个进程的 `flag[j]`，如果 `flag[j]` 为 `true`，意味着另一个进程也准备进入临界区，那么它会查看 `turn` 是否指向自己。如果 `turn == j`，表示轮到进程 `j` 进入临界区，那么进程 `i` 就会暂停，等待进程 `j` 执行完后再进入。只有当 `turn == i`，或者 `flag[j] == false` 时，进程 `i` 才能进入临界区执行它的操作。

这种机制确保了互斥性，因为在任意时刻，只可能有一个进程进入临界区。当一个进程进入临界区并完成操作后，它会将自己的 `flag[i]` 设置为 `false`，表示它退出临界区，其他进程就可以重新尝试进入。

Peterson 算法保证了多个重要的特性。首先是**互斥性**，即在任意时刻只有一个进程能够进入临界区，从而避免了数据竞争的发生。其次是**无死锁性**，即两个进程不会因为相互等待而导致死锁。再者，**有限等待**也得到保证，每个进程最终都会在有限的时间内进入临界区，不会被永久阻塞。此外，算法还保证了进程间的**公平性**，即两个进程能够交替进入临界区，而不会导致某一个进程被长时间饿死。

尽管 Peterson 算法在理论上提供了一种有效的解决方案，但它在实际应用中受到一些限制。例如，它假设硬件提供原子性的读写操作，并且依赖于忙等待（busy waiting）机制，这在多处理器系统中可能导致性能问题。此外，随着进程数目的增加，Peterson 算法会变得不再适用，因为它仅适用于两个进程的情形，因此在更复杂的系统中，需要其他更加高效和通用的同步机制。

1.4 逻辑地址与物理地址

逻辑地址与物理地址的转换是本次实验的另一个重点。

逻辑地址和物理地址是计算机系统中与内存相关的两个重要概念，它们在操作系统和计算机体系结构中扮演着至关重要的角色。逻辑地址和物理地址之间的关系通过内存管理单元（MMU）来实现，是现代操作系统和硬件设计的基础。

逻辑地址，也叫做虚拟地址，是由程序在运行过程中生成的地址。当一个程序运行时，它并不知道它所使用的内存是如何在实际的物理内存中分配的，程序只是使用它所认为的“虚拟内存地址”。这些地址是程序在编译时或运行时生成的，通常是相对独立的，不直接与物理内存中的地址关联。操作系统和硬件会利用虚拟内存技术，通过一种叫做**地址映射**的机制，将这些虚拟地址转换成实际的物理地址。

物理地址是实际内存硬件中存储数据的地址，是计算机系统中内存的真实地址。物理内存指的是计算机中的实际内存芯片，它直接存储着所有的程序和数据。每个内存单元都有一个唯一的物理地址，CPU 通过物理地址来访问这些内存单元。

在早期的计算机系统中，程序的逻辑地址和物理地址是相同的，也就是说，程序直接使用物理地址来访问内存。但是，随着操作系统和硬件的发展，尤其是在多任务操作系统和虚拟内存的引入后，程序使用的逻辑地址和实际的物理地址之间就产生了区分。这种区分使得操作系统可以在一个进程的虚拟地址空间和另一个进程的虚拟地址空间之间进行隔离，提高了系统的安全性和效率。

逻辑地址到物理地址的转换是通过内存管理单元 (MMU) 来完成的。MMU 是计算机系统中的一个硬件部件，它负责将程序生成的虚拟地址转换成物理地址。转换的过程通常通过页表 (Page Table) 来实现。操作系统将虚拟内存划分为若干个**页** (Page)，而物理内存则划分为若干个**帧** (Frame)。页表记录了虚拟页和物理帧之间的映射关系。当程序访问某个虚拟地址时，MMU 会通过页表找到对应的物理地址，从而实现对实际内存的访问。

这种虚拟地址到物理地址的映射不仅提供了地址空间的隔离，防止了进程间直接访问彼此的内存，保护了系统的稳定性和安全性；同时，它也提供了内存的抽象和灵活性，使得操作系统可以更高效地管理内存。例如，操作系统可以将物理内存的不同部分分配给不同的进程，同时允许不同的进程在虚拟内存空间中拥有各自独立的地址空间，从而使得每个进程的地址空间看起来像是连续的、独立的，这种虚拟内存的概念极大地提高了系统的内存使用效率。

虚拟地址与物理地址的映射并不是一成不变的。操作系统，例如 Linux，可以通过一些技术，如**地址空间布局随机化** (ASLR) 来随机化虚拟地址空间的布局，从而提高系统的安全性。页面置换算法也是一种操作系统管理虚拟内存的重要方式，它决定了哪些页面应该被保留在内存中，哪些页面应该被写回磁盘，从而优化内存的使用。

2 实验环境

- 操作系统: ubuntu 20.04 LTS (Focal Fossa)
- 内核版本: Linux kernel 5.15
- 编程语言: c++23
- 编译器: GCC 14.2
- 构建工具: xmake v2.9.7+20241219
- 文档构建工具: doxygen 1.12.0

3 设计思路

3.1 共享内存的创建和初始化

为了让两个进程能够访问同一内存区域，我们使用了共享内存。共享内存是进程间通信（IPC）的一种方式，它允许多个进程访问同一块内存区域。在 Linux 系统中，可以使用 `mmap` 来创建和映射共享内存。通过使用 `MAP_ANONYMOUS` | `MAP_SHARED` 标志来创建匿名共享内存，这意味着内存区域没有与文件关联，并且共享给所有进程。

初始化时，我们将共享内存中的 `turn` 和 `flag` 变量设置为初始值，`turn` 设置为 0（表示进程 0 首先进入临界区），而 `flag` 数组初始化为 `false`（表示两进程都不想进入临界区）。

3.2 Peterson 算法的实现

Peterson 算法通过两个进程共享的变量 `turn` 和 `flag[]` 来协调进程对临界区的访问。每个进程在进入临界区之前，都将自己的 `flag[i]` 设置为 `true`，并将 `turn` 设置为另一个进程的编号，表示自己愿意让对方先进入临界区。然后，进程检查另一个进程的 `flag[j]` 和 `turn`，确保在对方希望进入时，它会等待。只有当对方不希望进入，或者轮到自己时，进程才能进入临界区。进入临界区后，进程会执行自己的任务，并在退出时将 `flag[i]` 设置为 `false`，表示它已经完成并不再需要访问临界区。

该代码中通过 `peterson_algorithm` 函数实现了每个进程的行为，其中包括请求进入临界区、执行临界区操作、退出临界区等步骤。该函数会循环执行 2 次，模拟进程反复请求进入临界区的情况。

伪代码如 algorithm 1 所示。

3.3 获取虚拟地址和物理地址

虚拟内存（Virtual Memory）机制将一个进程的地址空间分为若干页（Page），通常每页大小为 4KB（用 `_SC_PAGESIZE` 获取）。操作系统通过页表维护虚拟地址到物理地址的映射关系。

3.4 虚拟地址结构

虚拟地址可以分解为两部分：

- 页面号（Page Number）：确定该虚拟地址属于哪一页。

Algorithm 1: Peterson Algorithm for Mutual Exclusion (for two processes)

Input: Two processes with shared variables $flag[0], flag[1]$ and $turn$

Output: Mutual exclusion for both processes

Function Peterson(i, j):

```

    while True do
        flag[i] ← True ;           // Process i wants to enter critical section
        turn ← j ;                 // Give priority to process j
        while flag[j] ∧ turn = j do
            ;                       // Busy waiting, process i waits until process j exits
        ;                           // Critical section begins
        Enter critical section of process i;
        Perform operations;
        Exit critical section of process i;
        flag[i] ← False ;          // Process i exits critical section

```

- **页内偏移量 (Offset)**: 表示地址在页内的具体位置。

设页面大小为 $PageSize$, 则:

- 高位 $VirtualAddress/PageSize$ 表示页面号。
- 低位 $VirtualAddress/PageSize$ 表示页内偏移量。

页表存储了虚拟页面与物理页帧的映射关系:

- 每个虚拟页面对应一个条目, 记录物理页帧号 (Page Frame Number, PFN)。
- 页表条目中包含的其他标志位 (例如有效位、读写权限位等) 用于控制访问权限。

当 CPU 访问虚拟地址时, 内存管理单元 (MMU) 会通过以下步骤进行地址转换:

- 提取虚拟地址的页面号, 查找页表中对应的物理页帧号。
- 将页帧号与页内偏移量组合, 得到物理地址。

`/proc/self/pagemap` 是 Linux 中一个特殊的设备文件, 详细解释可以看[Linux 官方文档](#), 文件为用户态提供了当前进程虚拟地址到物理地址的映射信息。它是一个连续的二进制文件, 按每页对应一个 64 位条目排列:

- 第 63 位: 页面是否驻留在物理内存中 (Page Present)。

- 第 0 至 54 位：物理页帧号（PFN）。

文件索引与虚拟页号直接对应，因此可以通过计算虚拟地址的页号找到相关信息。

映射过程可以分为以下几步：

1. 计算虚拟页号

$$\text{Page Number} = \text{Virtual Address} / \text{Page Size}$$

2. 定位页表条目

每个页表条目占用 64 位（8 字节），因此可以通过偏移计算找到虚拟页号对应的条目位置：

$$\text{Offset} = \text{Page Number} \times 8$$

3. 读取条目信息

打开 `/proc/self/pagemap` 文件并跳转到计算出的偏移位置，读取 64 位数据。

4. 解析条目信息

解析数据中的位信息：

- 第 63 位为页面存在标志。
- 第 0 至 54 位为物理页帧号。

5. 计算物理地址

然后便可以通过计算得到物理地址：

$$\text{Physical Address} = (\text{PFN} \times \text{Page Size}) + (\text{Virtual Address} \& (\text{Page Size} - 1))$$

其中：

- PFN：物理页帧号。
- $\text{Virtual Address} \& (\text{Page Size} - 1)$ ：页内偏移量。

转化成伪代码如 algorithm 2 所示。

Algorithm 2: Get Physical Address and Verify**Input:** Virtual address *vaddr***Output:** Physical address *paddr***Function** `virtual_to_physical_address(vaddr):`

```

    pagemap_file  $\leftarrow$  "/proc/self/pagemap";
    fd  $\leftarrow$  open(pagemap_file, O_RDONLY|O_CLOEXEC);
    if fd < 0 then
        return Failure, "Unable to open pagemap file"
    page_size  $\leftarrow$  sysconf(_SC_PAGE_SIZE);
    page_number  $\leftarrow$   $\frac{vaddr}{page\_size}$ ;
    offset  $\leftarrow$  vaddr mod page_size;
    file_offset  $\leftarrow$  page_number  $\times$  8;
    pagemap_entry  $\leftarrow$  read(fd, file_offset);
    if pagemap_entry & (1 << 63) = 0 then
        return Failure, "Page not present"
    pfn  $\leftarrow$  pagemap_entry & ((1 << 54) - 1);
    physical_addr  $\leftarrow$  (pfn  $\times$  page_size) + offset;
    return physical_addr

```

3.5 物理地址验证

在获得物理地址后，代码通过 `verify_physical_address` 函数来验证该物理地址是否正确。验证过程通过 `mmap` 和 `/dev/mem` 文件实现，详细文档见 [dev/mem](#) 与 [mmap](#)。`/dev/mem` 也是 Linux 中一个特殊的设备文件，允许访问物理内存，正因如此，他的需求权限更高，并且默认屏蔽，前面提到的 `/proc/self/pagemap` 只需要 `root` 权限即可，该文件需要修改内核开启，否则只能访问前 1MB 的数据。通过在物理地址处读取数据，我们可以确认该物理地址是否正确。

- 通过 `/dev/mem` 文件映射物理内存，读取物理地址对应位置的数据。
- 比较读取的值与期望值是否一致，从而验证物理地址的正确性。

3.6 进程间同步和协作

在主进程和子进程间通过共享内存传递数据，确保两进程能在相同的内存区域进行同步。在主进程中，首先通过 `fork()` 创建子进程，然后主进程和子进程各自执行 `peterson_algorithm` 函数，模拟并发访问临界区。

两进程在访问临界区时，通过 Peterson 算法进行同步，确保在任一时刻，只有一个进程能够进入临界区执行操作。为了避免 CPU 占用过高，进程在等待时使用了 `sleep(100)`，具体实现时，采

用了线程安全的休眠方式，给 CPU 释放一定的时间，以减少不必要的忙等待。

4 实现方法：修改 Linux 内核

由于在现代 Linux 系统（如 Ubuntu）中，出于安全性和系统稳定性的考虑，对 `/dev/mem` 的访问受到严格限制。如果需要允许用户空间程序直接访问 `/dev/mem`，需要通过重新配置并编译 Linux 内核来实现。考虑到没有给出实验说明，下面将详细给出操作过程，均为本人查看相关文档原创而来，建议老师可以考虑增加说明文档。

4.1 获取内核源代码

首先，需要获取 Linux 内核的源代码，以便进行更改。为了确保不引入其他问题，我们在对应的版本代码上进行修改，首先使用 `uname` 命令查看自己的内核版本，我们的内核版本为 5.15：

1 `uname -r`

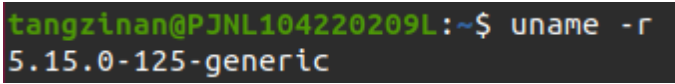


图 1: `uname` 命令查看内核版本

由于官方网站仅提供部分内核下载，且下载速度较慢，在[镜像网站](#)上找到对应的版本下载。下载

linux-5.15.97.tar.sign	03-Mar-2023 11:01	991
linux-5.15.97.tar.xz	03-Mar-2023 11:01	121M
linux-5.15.98.tar.gz	03-Mar-2023 14:23	186M
linux-5.15.98.tar.sign	03-Mar-2023 14:23	991
linux-5.15.98.tar.xz	03-Mar-2023 14:23	121M
linux-5.15.99.tar.gz	10-Mar-2023 08:48	186M
linux-5.15.99.tar.sign	10-Mar-2023 08:48	991
linux-5.15.99.tar.xz	10-Mar-2023 08:48	121M
linux-5.15.tar.gz	31-Oct-2021 21:54	186M
linux-5.15.tar.sign	31-Oct-2021 21:54	985
linux-5.15.tar.xz	31-Oct-2021 21:54	116M
linux-5.16.1.tar.gz	16-Jan-2022 08:18	188M
linux-5.16.1.tar.sign	16-Jan-2022 08:18	989
linux-5.16.1.tar.xz	16-Jan-2022 08:18	122M
linux-5.16.10.tar.gz	16-Feb-2022 12:13	188M

图 2: 从镜像网站下载 Linux 内核源代码

后的文件放入 `/usr/src/` 目录下。

1 `sudo mv linux-5.15.tar.gz /usr/src/ #`

解压并进入通过包管理器下载的内核源代码：

```
1  cd /usr/src/  
2  sudo tar -xf linux-5.15.tar.gz #  
3  cd linux-5.15 #
```

随后确保已安装编译内核所需的依赖包：

```
1  sudo apt update  
2  sudo apt upgrade  
3  sudo apt-get install build-essential libncurses-dev bison flex libssl-dev  
   ↪ libelf-dev dwarves zstd
```

这些软件包包括编译工具链和库文件，用于配置和编译内核。

4.2 配置内核

首先需要复制系统当前的配置文件，否则新的内核可能不会启用当前系统中使用的某些硬件或驱动程序，如果某些功能或驱动程序没有被启用，可能会导致系统无法正常启动或部分硬件无法工作。

```
1  cp /boot/config-$(uname -r) .config
```

使用 `make menuconfig` 进行可视化配置，`menuconfig` 是一个基于文本的配置工具，可以通过图形界面选择和配置内核选项。

```
1  sudo make menuconfig
```

下面便是关键的内核修改配置。

为了便于区分自己修改的内核，首先选中 `General setup` → `Local version` 回车，设置自定义内核版本后缀，如 `-devmem`。

选中 `Kernel hacking` → `Filter access to /dev/mem` 选项，空格取消选择，以禁用 `CONFIG_STRICT_DEVMEM` 选项。这是一项安全功能，限制了用户空间程序对 `/dev/mem` 的访问，使得用户空间只能访问 1MB 的物理内存。由于不同版本的内核配置选项可能有所不同，具体选项可能会有所不同，但一般都会有类似的选项来控制对 `/dev/mem` 的访问。可以通过搜索关键字来查找选项，

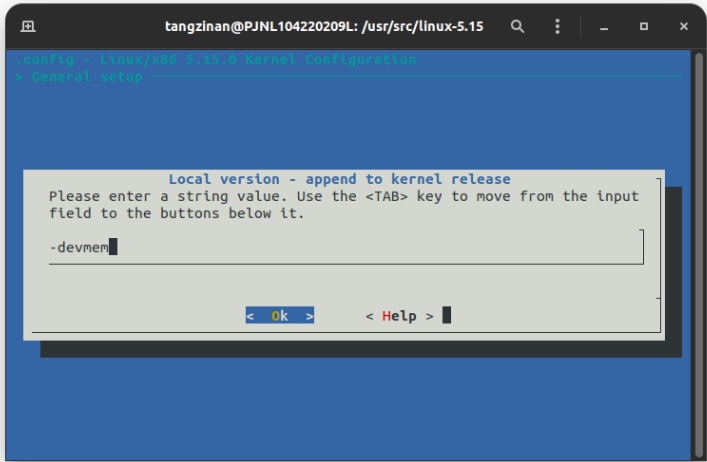


图 3: 设置自定义内核版本后缀

输入 / 后输入关键字进行搜索，可以看到对应的路径。修改后选择 **Save** 保存配置至 `.config` 文件。

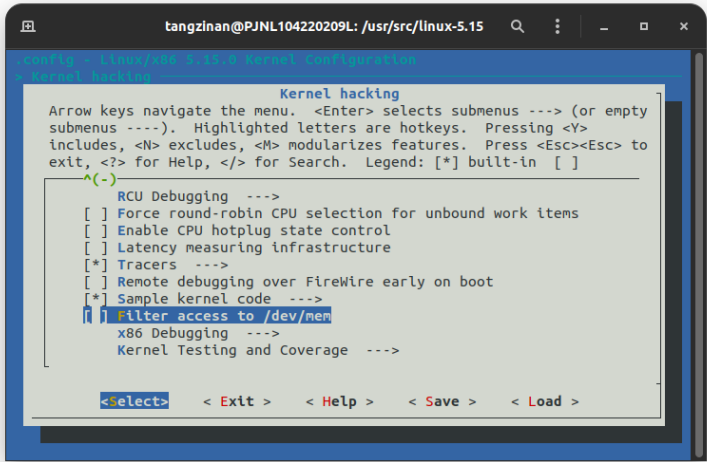


图 4: 取消选项

此外，在较新的版本中，`HARDENED_USERCOPY` 和 `HAVE_HARDENED_USERCOPY_ALLOCATOR` 选项会导致 `CONFIG_STRICT_DEVMEM` 开启，路径为 `Security options` → `Harden memory copies between kernel and userspace`，也需要确认这两个选项禁用（另外一个一般是默认禁用，可以用下面查看文件的方法检查）。

如果在 `menuconfig` 中找不到这些选项，可以直接编辑 `.config` 文件，将 `CONFIG_STRICT_DEVMEM`

设置为 `n`，并将 `CONFIG_HARDENED_USERCOPY` 和 `CONFIG_HAVE_HARDENED_USERCOPY_ALLOCATOR` 设置为 `n`，或者干脆将这些选项注释掉。

可以使用 `vim` 或者 `nvim` 进行修改，推荐使用 `nvim` 更新也更快，如果没有下载，使用 `sudo apt install` 下载，输入 `/` 即可进行查找。`vim` 的相关操作可见[Vim 从入门到精通](#)。

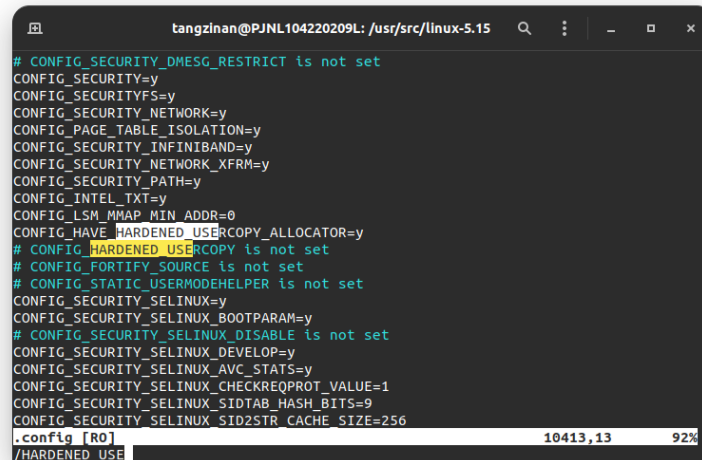


图 5: 直接编辑配置文件

修改完配置后，还需要设置关闭启用模块签名，以便进行编译。

```
1 scripts/config --disable SYSTEM_TRUSTED_KEYS
2 scripts/config --disable SYSTEM_REVOCATION_KEYS
```

因为我们自己修改的内核是没有经过签名的，那么在启用了 `SYSTEM_TRUSTED_KEYS` 和 `SYSTEM_REVOCATION_KEYS` 的情况下，内核会拒绝加载它们。禁用这些选项可以避免签名检查，允许加载没有签名的模块。

4.3 编译和安装自定义内核

首先清理之前的编译结果，避免对后续编译产生影响。

```
1 sudo make clean
```

使用以下命令并行编译 (`-j$(nproc)` 会根据的 CPU 核心数并行编译，加快编译速度)：

```
1 sudo make -j$(nproc)
```

编译模块：

```
1 sudo make modules
```

编译内核映像：

```
1 sudo make bzImage
```

编译完成后，需要安装内核映像和模块到系统中。

```
1 sudo make modules_install
```

这将把编译好的模块安装到 `/lib/modules/<kernel-version>/` 目录下。

安装内核映像。

```
1 sudo make install
```

此命令会将内核映像（如 `vmlinuz-<version>`）、系统映像和内核配置文件安装到 `/boot/` 目录，并自动更新引导加载器（如 GRUB）。

4.4 重启并更换内核

为了防止不显示 GRUB 菜单，我们先修改 GRUB 配置。

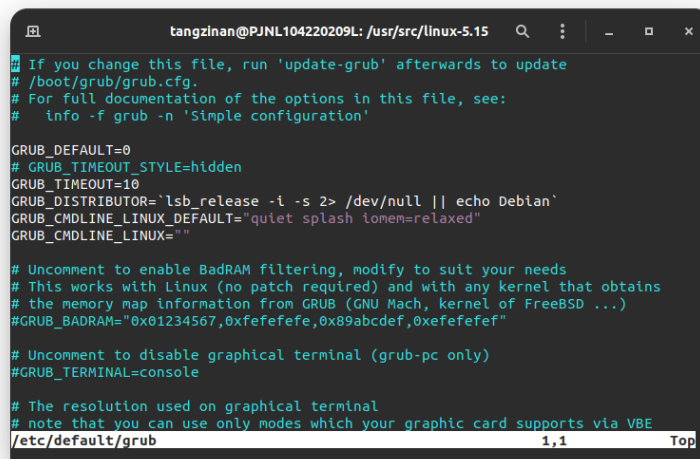
```
1 sudo nvim /etc/default/grub
```

将 `GRUB_TIMEOUT` 设置为一个较大的值，如 10 秒，以确保 GRUB 菜单显示，并注释掉 `GRUB_HIDDEN_TIMEOUT`。

手动更新 GRUB 配置：

```
1 sudo update-grub
```

随后重启系统：



```

tangzinan@PJNL104220209L: /usr/src/linux-5.15
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
# GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=10
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash tomem=relaxed"
GRUB_CMDLINE_LINUX=""

# Uncomment to enable BadRAM filtering, modify to suit your needs
# This works with Linux (no patch required) and with any kernel that obtains
# the memory map information from GRUB (GNU Mach, kernel of FreeBSD ...)
#GRUB_BADRAM="0x01234567,0xfefefefe,0x89abcdef,0xefefefef"

# Uncomment to disable graphical terminal (grub-pc only)
#GRUB_TERMINAL=console

# The resolution used on graphical terminal
# note that you can use only modes which your graphic card supports via VBE
/etcd/default/grub 1,1 Top

```

图 6: 编辑 GRUP 配置文件

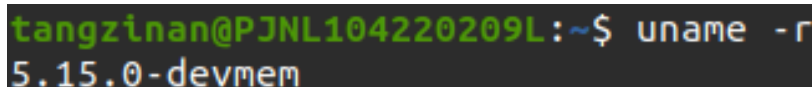
```
1 sudo reboot
```

在启动时按住 Shift 键（对于 BIOS 系统，常见于虚拟机）或 Esc 键（对于 UEFI 系统）以显示菜单。在系统启动时，GRUB 菜单将显示多个内核选项。选择 **Advanced options for Ubuntu**，然后选择新安装（此前我们已经设置过后缀）的内核版本，启动系统。

4.5 验证内核配置

有时候会因为 Nvidia 掉驱动导致无法正常启动页面，在 **Advanced options for Ubuntu** 菜单中选择对应 **recovery mode**，首先选择 **network** 连接网络，再选择 **dpkg** 修复包即可。

进入系统后，再次 `uname -r` 查看内核版本，确保已经切换到新内核。



```

tangzinan@PJNL104220209L:~$ uname -r
5.15.0-devmem

```

图 7: 已经切换到新内核

5 系统框架及关键代码

5.1 main() 函数

main() 函数是程序的入口。

5.1.1 分配共享内存

查看文档，mmap 函数可以被用来创建共享内存，其借口定义如下：

```
1 void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t  
↪ offset);
```

1. void *addr

- **含义：**建议的映射起始地址。
- **作用：**
 - 如果为 `NULL`，则表示让内核自动选择映射地址。
 - 如果不为 `NULL`，内核会尝试将映射放在指定地址（可能会失败）。
- **注意：**
 - 通常设置为 `NULL` 以让内核选择最合适的地址。

2. size_t length

- **含义：**映射的内存区域大小（以字节为单位）。
- **作用：**
 - 指定映射区域的长度。
 - 映射的区域会从 `offset` 开始，向后延续 `length` 字节。
- **注意：**
 - 必须为系统页大小（通常是 4KB）的倍数，否则会向上对齐到页大小。

3. int prot

- **含义：**内存保护属性。

- **取值：**可以是以下宏的组合（通过按位或 | 连接）：
 - PROT_READ：可读。
 - PROT_WRITE：可写。
 - PROT_EXEC：可执行。
 - PROT_NONE：不可访问。
- **示例：**
 - 只读：PROT_READ
 - 可读写：PROT_READ | PROT_WRITE

4. int flags

- **含义：**映射类型和属性标志。
- **取值：**以下标志可以单独使用或组合：
 - 映射类型（必选之一）：
 - * MAP_SHARED：共享映射。对内存区域的修改会写回到文件，并对其他进程可见。
 - * MAP_PRIVATE：私有映射。对内存区域的修改不会影响原文件或其他进程。
 - 其他标志（可选）：
 - * MAP_ANONYMOUS：匿名映射，不与文件关联（fd 必须为 -1）。
 - * MAP_FIXED：强制使用 addr 指定的地址。如果地址不可用，会失败。
 - * MAP_POPULATE：预先加载页表。

5. int fd

- **含义：**要映射的文件描述符。
- **作用：**
 - 指定需要映射的文件。
 - 如果使用匿名映射（MAP_ANONYMOUS），必须设置为 -1。
- **注意：**
 - 文件描述符必须支持读取或读写（根据 prot）。

6. off_t offset

- **含义：**映射区域在文件中的偏移量。
- **作用：**
 - 指定映射的起始位置，必须是系统页大小（通常是 4KB）的倍数。
- **注意：**
 - 如果不是页大小的倍数，映射会失败。

返回值：

- 成功：返回指向映射区域的指针。
- 失败：返回 `MAP_FAILED`（即 `(void *)-1`），并设置 `errno`。

```
1 int* shared_mem = static_cast<int*>(mmap(nullptr, sizeof(int) * 3, PROT_READ |  
↪ PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0));
```

使用 `mmap` 创建了一个匿名共享内存区域，用于在父子进程间共享数据。分配了 3 个 `int` 大小的区域，用于存放 `flag` 和 `turn`。

5.1.2 初始化共享变量

```
1 volatile int* flag = reinterpret_cast<volatile int*>(shared_mem);  
2 volatile int* turn = reinterpret_cast<volatile int*>(shared_mem + 2);  
3 flag[0] = 0;  
4 flag[1] = 0;  
5 *turn = 0;
```

初始化两个进程的 `flag` 值为 0，表示开始时均无意进入临界区，`turn` 设置为 0 表示在冲突时优先让进程 0 先尝试进入。在多线程或多进程共享内存中如果使用普通 `int` 可能被编译器或 CPU 缓存重排，从而破坏算法预期。因此使用 `volatile` 限定符，强制编译器不优化相关内存访问。

5.1.3 创建子进程

```
1 pid_t pid = fork();
```

使用 `fork()` 创建子进程，形成两个进程，父进程 `pid>0`，使用 `i=0`，子进程 `pid=0`，使用 `i=1`。父进程在完成自身的临界区访问后使用 `wait(nullptr)` 等待子进程结束，然后使用 `munmap` 解除内存映射，释放资源。

5.2 Peterson 算法： `peterson_algorithm()` 函数

```

1  for (int count = 0; count < 2; count++) {
2      flag[i] = 1;      //          i
3      *turn = j;      //      turn
4
5      //          (busy-wait)
6      while ((flag[j] != 0) && (*turn == j)) {
7          //          turn
8      }
9
10     // ----          ----
11     std::cout << "Process " << i << " entered critical section (iteration " <<
        << count << ")\n";
12
13     //          turn
14     verify_physical_address(turn, i);
15
16     //
17     std::this_thread::sleep_for(std::chrono::seconds(1));
18     std::cout << "Process " << i << " leaving critical section (iteration " <<
        << count << ")\n\n";
19
20     //
21     flag[i] = 0;
22
23     //

```

```
24     std::this_thread::sleep_for(std::chrono::seconds(1));  
25 }
```

实现方法与前文所述的伪代码一致，通过 `flag` 和 `turn` 变量实现进程间的同步。值得注意的是，由于需要验证物理地址正确，正好可以将临界区行为设置为验证物理地址。此外，我们没有使用普通的 `sleep()` 函数，而是使用了线程安全的 `std::this_thread::sleep_for()` 函数，该函数专为 C++ 线程设计，明确表示让当前线程暂停执行，语义更加清晰，以避免在多线程环境下可能出现的问题。

5.3 获取物理地址：virtual_to_physical_address() 函数

实现将虚拟地址转换为物理地址的核心逻辑。

首先确定系统页面的大小

```
1 auto page_size = static_cast<uint64_t>(sysconf(_SC_PAGESIZE));
```

根据虚拟地址与页面大小计算在 `pagemap` 文件中的偏移：

```
1 uint64_t offset = (vaddr / page_size) * sizeof(uint64_t);
```

打开 `pagemap` 文件并定位：

```
1 FILE* pagemap = fopen("/proc/self/pagemap", "rbe");  
2 fseek(pagemap, static_cast<int64_t>(offset), SEEK_SET);
```

读取对应的 64 位条目 `e`：

```
1 fread(&e, sizeof(uint64_t), 1, pagemap);
```

条目 `e` 中包含该虚拟页对应的物理页帧号 (PFN) 以及其他标志位。当 `e` 的最高位 (第 63 位) 为 1 时，表示此页已映射到物理内存。

提取 PFN 并计算物理地址：

```
1 uint64_t pfn = e & ((1ULL << 54U) - 1);  
2 paddr = pfn * page_size + (vaddr & (page_size - 1));
```

`pfn` 是物理页帧号，通过 `pfn * page_size` 得到物理页的起始地址，然后加上 `vaddr` 在页面内的偏移量得到完整的物理地址。

值得注意的是，在实现时我们使用了 `static_cast` 以及 `reinterpret_cast` 等 C++ 强制类型转换，编译器可以严格检查转换是否合法。例如，它可以在编译期捕捉明显错误。这样做更现代化、符合类型安全且可读性更高，能够明确表达意图，并有效减少不安全的类型转换带来的隐患。为了保证跨平台一致性，使用 `uint64_t` 而不是 `long` 来定义相关变量。

5.4 验证物理函数正确：verify_physical_address() 函数

通过验证共享内存中 `turn` 的物理地址对应的值与当前进程逻辑访问该地址的值是否一致，从而说明进程间通过共享内存访问同一物理地址。

首先获取 `turn` 的虚拟地址并转换为物理地址

```
1 auto vaddr = reinterpret_cast<uintptr_t>(turn);
2 uintptr_t paddr = virtual_to_physical_address(vaddr);
```

若转换成功 (`paddr != 0`)，则打开 `/dev/mem`，通过 `lseek` 定位到 `paddr` 对应的物理位置，然后 `read` 读取该处的内存值。

```
1 int memfd = open("/dev/mem", O_RDONLY | O_CLOEXEC);
2 // ...
3 ssize_t rd = read(memfd, &val_at_paddr, sizeof(val_at_paddr));
```

将从物理地址读取到的 `val_at_paddr` 与逻辑读取的 `*turn` 值进行比较，并根据是否相等输出验证结果。

6 模块流程图

由 doxygen 自动生成。



图 8: 模块流程图

7 测试

在任一时刻只有一个进程进入临界区，此外，通过物理地址直接访问的 `turn` 值与预期值一致，说明了物理地址的正确性。

```
● tangzinan@PJNL104220209L:~/operating/operating$ sudo xmake run --root
[sudo] tangzinan 的密码:
Process 0 entered critical section (iteration 0)
Process 0: Turn logical address: 0x7f11cf946008
Process 0: Turn physical address: 0x382131008
Process 0: Value stored at turn (logical): 0x0
Process 0: Value read from physical address: 0x0
Process 0: Verification successful: The value matches.
Process 0 leaving critical section (iteration 0)

Process 1 entered critical section (iteration 0)
Process 1: Turn logical address: 0x7f11cf946008
Process 1: Turn physical address: 0x382131008
Process 1: Value stored at turn (logical): 0x0
Process 1: Value read from physical address: 0x0
Process 1: Verification successful: The value matches.
Process 1 leaving critical section (iteration 0)

Process 0 entered critical section (iteration 1)
Process 0: Turn logical address: 0x7f11cf946008
Process 0: Turn physical address: 0x382131008
Process 0: Value stored at turn (logical): 0x1
Process 0: Value read from physical address: 0x1
Process 0: Verification successful: The value matches.
Process 0 leaving critical section (iteration 1)

Process 1 entered critical section (iteration 1)
Process 1: Turn logical address: 0x7f11cf946008
Process 1: Turn physical address: 0x382131008
Process 1: Value stored at turn (logical): 0x0
Process 1: Value read from physical address: 0x0
Process 1: Verification successful: The value matches.
Process 1 leaving critical section (iteration 1)
```

图 9: 程序运行结果

A 程序文档 by doxygen

Address Transformation

Generated on Mon Dec 23 2024 16:37:05 for Address Transformation by Doxygen 1.12.0

Mon Dec 23 2024 16:37:05

1 File Index	1
1.1 File List	1
2 File Documentation	3
2.1 src/main.cpp File Reference	3
2.1.1 Function Documentation	4
2.1.1.1 main()	4
2.1.1.2 peterson_algorithm()	4
2.1.1.3 verify_physical_address()	5
2.1.1.4 virtual_to_physical_address()	6
Index	7

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

src/[main.cpp](#) 3

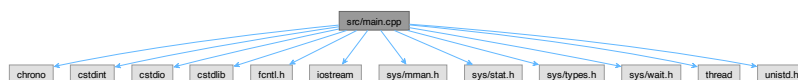
Chapter 2

File Documentation

2.1 src/main.cpp File Reference

```
#include <chrono>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <fcntl.h>
#include <iostream>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <thread>
#include <unistd.h>
```

Include dependency graph for main.cpp:



Functions

- `uintptr_t virtual_to_physical_address (uintptr_t vaddr)`
Converts a virtual address to a physical address by reading `/proc/self/pagemap`.
- `void verify_physical_address (const volatile int *turn, int i)`
Verifies that the physical address contents match the logical value of 'turn'.
- `void peterson_algorithm (int i, volatile int *flag, volatile int *turn)`
The main routine for each process in Peterson's Algorithm.
- `int main ()`
Program entry point.

2.1.1 Function Documentation

2.1.1.1 main()

```
int main ()
```

Program entry point.

Allocates shared memory for the Peterson Algorithm's flags and turn variable. Forks a child process, assigning roles (0 for parent, 1 for child). Both processes execute [peterson_algorithm\(\)](#). The parent waits for the child to finish and then releases the shared memory.

Returns

EXIT_SUCCESS on successful execution, or EXIT_FAILURE on errors.

Here is the call graph for this function:



2.1.1.2 peterson_algorithm()

```
void peterson_algorithm (
    int i,
    volatile int * flag,
    volatile int * turn)
```

The main routine for each process in Peterson's Algorithm.

This function implements two iterations of the following:

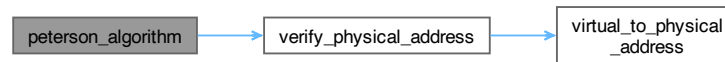
- Set the flag to indicate that this process wants to enter the critical section.
- Set turn to the other process.
- Busy-wait until it is safe to enter the critical section.
- Enter the critical section, verify the physical address mapping of 'turn'.
- Sleep for a moment to simulate some work in the critical section.
- Exit the critical section, reset the flag.
- Sleep to yield time to the other process.

Parameters

i	The process identifier (0 for parent, 1 for child).
flag	Pointer to the array of two flags (flag[0], flag[1]).

turn	Pointer to the shared 'turn' variable.
------	--

Here is the call graph for this function:



Here is the caller graph for this function:



2.1.1.3 verify_physical_address()

```
void verify_physical_address (
    const volatile int * turn,
    int i)
```

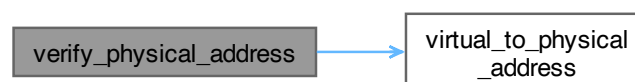
Verifies that the physical address contents match the logical value of 'turn'.

This function prints both the logical (virtual) and physical addresses of 'turn'. It then attempts to read the value from `/dev/mem` at the physical address to confirm that it matches the current value of 'turn'. Note that this step requires privileges to open `/dev/mem`.

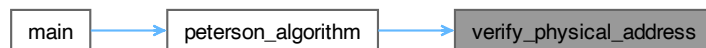
Parameters

turn	Pointer to the shared 'turn' variable.
i	The process identifier (0 for parent, 1 for child).

Here is the call graph for this function:



Here is the caller graph for this function:



2.1.1.4 virtual_to_physical_address()

```
uintptr_t virtual_to_physical_address (  
    uintptr_t vaddr)
```

Converts a virtual address to a physical address by reading `/proc/self/pagemap`.

This function reads the pagemap entry corresponding to the given virtual address, extracts the Page Frame Number (PFN) if present, and constructs the physical address.

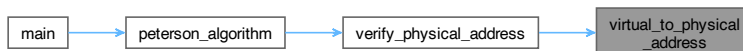
Parameters

vaddr	The virtual address to be converted.
-------	--------------------------------------

Returns

The corresponding physical address, or 0 on failure.

Here is the caller graph for this function:



Index

- main
 - main.cpp, [4](#)
- main.cpp
 - main, [4](#)
 - peterson_algorithm, [4](#)
 - verify_physical_address, [5](#)
 - virtual_to_physical_address, [6](#)
- peterson_algorithm
 - main.cpp, [4](#)
- src/main.cpp, [3](#)
- verify_physical_address
 - main.cpp, [5](#)
- virtual_to_physical_address
 - main.cpp, [6](#)