# lab4

## 编程题

### 1.使用sbrk,mmap,munmap,mprotect内存相关系统调用的linux应用程序。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#define SIZE 4096

int main() {
    // 使用sbrk分配内存
    printf("使用sbrk：\n");
    void* sbrk_ptr = sbrk(0); // 获取当前断点位置
    printf("初始断点值：%p\n", sbrk_ptr);

    sbrk(SIZE); // 分配SIZE字节的内存
    void* sbrk_new_ptr = sbrk(0); // 获取新的断点位置
    printf("sbrk分配后的新断点值：%p\n", sbrk_new_ptr);

    // 使用mmap分配内存
    printf("\n使用mmap：\n");
    void* mmap_ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE |
MAP_ANONYMOUS, -1, 0); // 使用mmap分配SIZE字节的内存
    if (mmap_ptr == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }
    printf("mmap分配的内存地址：%p\n", mmap_ptr);

    // 使用mprotect更改内存保护属性
    printf("\n使用mprotect：\n");
    if (mprotect(mmap_ptr, SIZE, PROT_READ) == -1) { // 更改内存保护属性为只读
        perror("mprotect");
        exit(1);
    }
    printf("内存保护属性已更改。不允许写入访问。\n");

    // 使用munmap释放内存
    printf("\n使用munmap：\n");
    if (munmap(mmap_ptr, SIZE) == -1) { // 解除内存映射并释放内存
        perror("munmap");
        exit(1);
    }
    printf("内存已解除映射并释放。\n");

    return 0;
```

```
}
```

这个示例程序展示了如何使用 `sbrk` 进行堆内存分配、`mmap` 进行内存映射、`mprotect` 更改内存保护属性以及 `munmap` 释放内存。请注意，`mmap` 和 `munmap` 通常用于对大块内存的映射和释放，而 `sbrk` 则更适用于对较小的内存块进行动态分配。

对 `sbrk`、`mmap`、`munmap` 和 `mprotect` 四个系统调用的解析：

1. `sbrk` 系统调用：
   - 函数原型：`void *sbrk(intptr_t increment);`
   - 功能：用于调整进程的堆空间大小。通过增加或减少堆的大小，可以动态地分配或释放内存。
   - 参数 `increment`：增量，表示要增加或减少的字节数。如果 `increment` 为正值，则堆的大小增加；如果为负值，则堆的大小减少。
   - 返回值：如果调用成功，则返回新分配的内存块的起始地址；如果调用失败，则返回 `(void *)-1`。
2. `mmap` 系统调用：
   - 函数原型：`void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
   - 功能：用于在进程的虚拟地址空间中创建一个新的映射区域，用于映射文件或匿名内存。
   - 参数 `addr`：指定映射区域的首选地址。如果为 `NULL`，则由系统自动选择合适的地址。
   - 参数 `length`：映射区域的长度（字节数）。
   - 参数 `prot`：指定内存保护标志，控制对映射区域的访问权限。
   - 参数 `flags`：指定映射区域的一些属性，如映射类型、共享方式等。
   - 参数 `fd`：文件描述符，表示要映射的文件。如果映射匿名内存而不是文件，则设置为 -1。
   - 参数 `offset`：文件偏移量，表示从文件中的哪个位置开始映射。对于匿名内存映射，通常设置为 0。
   - 返回值：如果调用成功，则返回新映射区域的起始地址；如果调用失败，则返回 `(void *)-1`。
3. `munmap` 系统调用：
   - 函数原型：`int munmap(void *addr, size_t length);`
   - 功能：用于解除一个或多个已映射的内存区域。
   - 参数 `addr`：要解除映射的起始地址。
   - 参数 `length`：要解除映射的长度（字节数）。
   - 返回值：如果调用成功，则返回 0；如果调用失败，则返回 -1。
4. `mprotect` 系统调用：
   - 函数原型：`int mprotect(void *addr, size_t len, int prot);`
   - 功能：用于更改指定内存区域的访问权限。
   - 参数 `addr`：要更改权限的内存区域的起始地址。
   - 参数 `len`：内存区域的长度（字节数）。
   - 参数 `prot`：指定新的内存保护标志，控制对内存区域的访问权限。
   - 返回值：如果调用成功，则返回 0；如果调用失败，则返回 -1。

## 2.扩展内核，实现Clock置换算法或二次机会置换算法。

Clock置换算法

```rust
fn pop(&mut self) -> Option<(VirtualPageNumber, FrameTracker)> {
    let mut i = 0; // 初始化循环索引
    let len = self.queue.len(); // 获取队列的长度
```

```
    loop {
        unsafe {
            let p = self.queue[i].2 as *mut PageTableEntry; // 获取PageTableEntry
的指针
            let mut flag = (*p).flags().clone(); // 克隆PageTableEntry的标志位

            if flag.contains(Flags::ACCESSED) { // 检查ACCESSED标志位是否被设置
                flag.set(Flags::ACCESSED, false); // 清除ACCESSED标志位
                (*p).set_flags(flag); // 更新PageTableEntry的标志位
            } else {
                let s = self.queue.remove(i); // 移除索引为i的元素
                return Some((s.0, s.1)); // 将移除的元素作为元组（vpn，FrameTracker）
返回
            }
            i = (i + 1) % len; // 增加循环索引，如果需要则回到0
        }
    }
}
fn push(&mut self, vpn: VirtualPageNumber, frame: FrameTracker, _entry: *mut
PageTableEntry) {
    self.queue.push((vpn, frame, _entry as usize)); // 将元素添加到队列的末尾
}
```

# 问答题

## 1.在使用高级语言编写用户程序的时候，手动用嵌入汇编的方法随机访问一个不在当前程序逻辑地址范围内的地址，比如向该地址读/写数据。该用户程序执行的时候可能会生什么?

可能会报出缺页异常.

## 2.用户程序在运行的过程中，看到的地址是逻辑地址还是物理地址?从用户程序访问某一个地址，到实际内存中的对应单元被读/写，会经过什么样的过程，这个过程中操作系统有什么作用？（站在学过计算机组成原理的角度）

逻辑地址。这个过程需要经过页表的转换，操作系统会负责建立页表映射。实际程序执行时的具体VA到PA的转换是在CPU的MMU之中进行的。

## 3.覆盖、交换和虚拟存储有何异同，虚拟存储的优势和挑战体现在什么地方?

它们都是采取层次存储的思路，将暂时不用的内存放到外存中去，以此来缓解内存不足的问题。

不同之处：覆盖是程序级的，需要程序员自行处理。交换则不同，由OS控制交换程序段。虚拟内存也由OS和CPU来负责处理，可以实现内存交换到外存的过程。

虚拟存储的优势:1.与段/页式存储完美契合，方便非连续内存分配。2.粒度合适，比较灵活。兼顾了覆盖和交换的好处：可以在较小粒度上置换；自动化程度高，编程简单，受程序本身影响很小。（覆盖的粒度受限于程序模块的大小，对编程技巧要求很高。交换粒度较大，受限于程序所需内存。尤其页式虚拟存储，几乎不受程序影响，一般情况下，只要置换算法合适，表现稳定、高效）3.页式虚拟存储还可以同时解决内存外碎片。提高空间利用率。

虚拟存储的挑战: 1.依赖于置换算法的性能。2.相比于覆盖和交换，需要比较高的硬件支持。3.较小的粒度在面临大规模的置换时会发生多次较小规模置换，降低效率。典型情况是程序第一次执行时的大量page fault，可配合预取技术缓解这一问题。

## 4.什么是局部性原理？为何很多程序具有局部性？局部性原理总是正确的吗？为何局部性原理为虚拟存储提供了性能的理论保证?

局部性分时间局部性和空间局部性（以及分支局部性）。局部性的原理是程序经常对一块相近的地址进行访问或者是对一个范围内的指令进行操作。局部性原理不一定是一直正确的。虚拟存储以页为单位，局部性使得数据和指令的访存局限在几页之中，可以避免页的频繁换入换出的开销，同时也符合TLB和cache的工作机制。

## 5.一条load指令，最多导致多少次页访问异常？尝试考虑较多情况。

考虑多级页表的情况。首先指令和数据读取都可能缺页。因此指令会有3次访存，之后的数据读取除了页表页缺失的3次访存外，最后一次还可以出现地址不对齐的异常，因此可以有7次异常。若考更加极端的情况，也就是页表的每一级都是不对齐的地址并且处在两页的交界处（straddle），此时一次访存会触发2次读取页面，如果这两页都缺页的话，会有更多的异常次数。

## 6.如果在页访问异常中断服务例程执行时，再次出现页访问异常，这时计算机系统（软件或硬件）会如何处理？这种情况可能出现吗?

我们实验的os在此时不支持内核的异常中断，因此此时会直接panic掉，并且这种情况在我们的os中这种情况不可能出现。像linux系统，也不会出现嵌套的page fault。

## 7全局和局部置换算法有何不同？分别有哪些算法?

全局页面置换算法：可动态调整某任务拥有的物理内存大小；影响其他任务拥有的物理内存大小。例如：工作集置换算法，缺页率置换算法。

局部页面置换算法：每个任务分配固定大小的物理页，不会动态调整任务拥有的物理页数量；只考虑单个任务的内存访问情况，不影响其他任务拥有的物理内存。例如：最优置换算法、FIFO置换算法、LRU置换算法、Clock置换算法。

## 8.简单描述OPT、FIFO、LRU、Clock、LFU的工作过程和特点 (不用写太多字，简明扼要即可)

OPT：选择一个应用程序在随后最长时间内不会被访问的虚拟页进行换出。性能最佳但无法实现。

FIFO：由操作系统维护一个所有当前在内存中的虚拟页的链表，从交换区最新换入的虚拟页放在表尾，最久换入的虚拟页放在表头。当发生缺页中断时，淘汰/换出表头的虚拟页并把从交换区新换入的虚拟页加到表尾。实现简单，对页访问的局部性感知不够。

LRU：替换的是最近最少使用的虚拟页。实现相对复杂，但考虑了访存的局部性，效果接近最优置换算法。

Clock：将所有有效页放在一个环形循环列表中，指针根据页表项的使用位（0或1）寻找被替换的页面。考虑历史访问，性能略差于但接近LRU。

LFU：当发生缺页中断时，替换访问次数最少的页面。只考虑访问频率，不考虑程序动态运行。

## 9.综合考虑置换算法的收益和开销，综合评判在哪种程序执行环境下使用何种算法比较合适?

> FIFO算法：在内存较小的系统中，FIFO 算法可能是一个不错的选择，因为它的实现简单，开销较小，但是会存在 Belady 异常。
>
> LRU算法：在内存容量较大、应用程序具有较强的局部性时，LRU 算法可能是更好的选择，因为它可以充分利用页面的访问局部性，且具有较好的性能。
>
> Clock算法：当应用程序中存在一些特殊的内存访问模式时，例如存在循环引用或者访问模式具有周期性时，Clock 算法可能会比较适用，因为它能够处理页面的访问频率。
>
> LFU算法：对于一些需要对内存访问进行优先级调度的应用程序，例如多媒体应用程序，LFU 算法可能是更好的选择，因为它可以充分考虑页面的访问频率，对重要性较高的页面进行保护，但是实现比较复杂。

## 10.Clock算法仅仅能够记录近期是否访问过这一信息，对于访问的频度几乎没有记录，如何改进这一点?

> 如果想要改进这一点，可以将Clock算法和计数器结合使用。具体做法是为每个页面设置一个计数器，记录页面在一段时间内的访问次数，然后在置换页面时，既考虑页面最近的访问时间，也考虑其访问频度。当待缓存对象在缓存中时，把其计数器的值加1。同时，指针指向该对象的下一个对象。若不在缓存中时，检查指针指向对象的计数器。如果是0，则用待缓存对象替换该对象；否则，把计数器的值减1，指针指向下一个对象。如此直到淘汰一个对象为止。由于计数器的值允许大于1，所以指针可能循环多遍才淘汰一个对象。

## 11.哪些算法有belady现象？思考belady现象的成因，尝试给出说明OPT和LRU等为何没有belady现象。

> FIFO算法、Clock算法。
>
> 页面调度算法可分为堆栈式和非堆栈式，LRU、LFU、OPT均为堆栈类算法，FIFO、Clock为非堆栈类算法，只有非堆栈类才会出现Belady现象。

## 12.什么是工作集？什么是常驻集？简单描述工作集算法的工作过程。

> 工作集为一个进程当前正在使用的逻辑页面集合，可表示为二元函数$W(t,Delta)$，t 为执行时刻，$Delta$ 称为工作集窗口，即一个定长的页面访问时间窗口，$W(t,Delta)$是指在当前时刻 t 前的 $Delta$ 时间窗口中的所有访问页面所组成的集合，$|W(t,Delta)|$为工作集的大小，即页面数目。

## 13.请列举 SV39 页`*`页表项的组成，结合课堂内容，描述其中的标志位有何作用 / 潜在作用?

> [63:54]为保留项，[53:10]为44位物理页号，最低的8位[7:0]为标志位。
>
> - V(Valid)：仅当位 V 为 1 时，页表项才是合法的；
> - R(Read)/W(Write)/X(eXecute)：分别控制索引到这个页表项的对应虚拟页面是否允许读/写/执行；
> - U(User)：控制索引到这个页表项的对应虚拟页面是否在 CPU 处于 U 特权级的情况下是否被允许访问；

> - A(Accessed)：处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被访问过；
> - D(Dirty)：处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被修改过。

## 14.请问一个任务处理 10G 连续的内存页面，需要操作的页表实际大致占用多少内存(给出数量级即可)?

大致占用`10G/512=20M`内存。

## 15.缺页指的是进程访问页面时页面不在页表中或在页表中无效的现象，此时 MMU 将会返回一个中断，告知操作系统：该进程内存访问出了问题。然后操作系统可选择填补页表并重新执行异常指令或者杀死进程。操作系统基于缺页异常进行优化的两个常见策略中，其一是 Lazy 策略，也就是直到内存页面被访问才实际进行页表操作。比如，一个程序被执行时，进程的代码段理论上需要从磁盘加载到内存。但是 操作系统并不会马上这样做，而是会保存 .text 段在磁盘的位置信息，在这些代码第一次被执行时才完成从磁盘的加载操作。 另一个常见策略是 swap 页置换策略，也就是内存页面可能被换到磁盘上了，导致对应页面失效，操作系统在任务访问到该页产生异常时，再把数据从磁盘加载到内存。

1. 哪些异常可能是缺页导致的？发生缺页时，描述与缺页相关的CSR寄存器的值及其含义。

- 答案： mcause 寄存器中会保存发生中断异常的原因，其中 Exception Code 为 12 时发生指令缺页异常，为 15 时发生 store/AMO 缺页异常，为 13 时发生 load 缺页异常。

CSR寄存器:

> - scause: 中断/异常发生时， CSR 寄存器 scause 中会记录其信息， Interrupt 位记录是中断还是异常， Exception Code 记录中断/异常的种类。
> - sstatus: 记录处理器当前状态，其中 SPP 段记录当前特权等级。
> - stvec: 记录处理 trap 的入口地址，现有两种模式 Direct 和 Vectored 。
> - sscratch: 其中的值是指向hart相关的S态上下文的指针，比如内核栈的指针。
> - sepc: trap 发生时会将当前指令的下一条指令地址写入其中，用于 trap 处理完成后返回。
> - stval: trap 发生进入S态时会将异常信息写入，用于帮助处理 trap ，其中会保存导致缺页异常的虚拟地址。

2. Lazy 策略有哪些好处？请描述大致如何实现Lazy策略?

- 答案：Lazy策略一定不会比直接加载策略慢，并且可能会提升性能，因为可能会有些页面被加载后并没有进行访问就被释放或替代了，这样可以避免很多无用的加载。分配内存时暂时不进行分配，只是将记录下来，访问缺页时会触发缺页异常，在`trap handler`中处理相应的异常，在此时将内存加载或分配即可。

3. swap 页置换策略有哪些好处？此时页面失效如何表现在页表项(PTE)上？请描述大致如何实现swap策略?

- 答案：可以为用户程序提供比实际物理内存更大的内存空间。页面失效会将标志位`v`置为`0`。将置换出的物理页面保存在磁盘中，在之后访问再次触发缺页异常时将该页面写入内存。

### 16.为了防范侧信道攻击，本章的操作系统使用了双页表。但是传统的操作系统设计一般采用单页表，也就是说，任务和操作系统内核共用同一张页表，只不过内核对应的地址只允许在内核态访问。(备注：这里的单/双的说法仅为自创的通俗说法，并无这个名词概念，详情见 KPTI )

1. 单页表情况下，如何控制用户态无法访问内核页面?

- 答案：将内核页面的 pte 的`U`标志位设置为0。

2. 相对于双页表，单页表有何优势?

- 答案：在内核和用户态之间转换时不需要更换页表，也就不需要跳板，可以像之前一样直接切换上下文。

3. 请描述：在单页表和双页表模式下，分别在哪个时机，如何切换页表?

- 答案：双页表实现下用户程序和内核转换时、用户程序转换时都需要更换页表，而对于单页表操作系统，不同用户线程切换时需要更换页表。

# 实践作业

## 重写 sys_get_time

引入虚存机制后，原来内核的 sys_get_time 函数实现就无效了。请你重写这个函数，恢复其正常功能。

## mmap 和 munmap 匿名映射

mmap 在 Linux 中主要用于在内存中映射文件，本次实验简化它的功能，仅用于申请内存。

请实现 mmap 和 munmap 系统调用，mmap 定义如下：

```
fn sys_mmap(start: usize, len: usize, prot: usize) -> isize
```

- syscall ID：222
- 申请长度为 len 字节的物理内存（不要求实际物理内存位置，可以随便找一块），将其映射到 start 开始的虚存，内存页属性为 prot
- 参数：

  start 需要映射的虚存起始地址，要求按页对齐len 映射字节长度，可以为 0prot：第 0 位表示是否可读，第 1 位表示是否可写，第 2 位表示是否可执行。其他位无效且必须为 0
- 返回值：执行成功则返回 0，错误返回 -1
- 说明：

  为了简单，目标虚存区间要求按页对齐，len 可直接按页向上取整，不考虑分配失败时的页回收。
- 可能的错误：

  start 没有按页大小对齐prot & !0x7 != 0 (prot 其余位必须为0)prot & 0x7 = 0 (这样的内存无意义)[start, start + len) 中存在已经被映射的页物理内存不足

munmap 定义如下：

```
fn sys_munmap(start: usize, len: usize) -> isize
```

- syscall ID：215
- 取消到 [start, start + len) 虚存的映射
- 参数和返回值请参考 mmap
- 说明：

  为了简单，参数错误时不考虑内存的恢复和回收。
- 可能的错误：

  [start, start + len) 中存在未被映射的虚存。