国科大操作系统研讨课任务书

RISC-V 版本

版本 2023

目录

Project 4
内存管理

1 实验说明

在之前的实验中，我们已经完成了进程的管理和通信，并实现了例外的处理，使得我们的操作系统可以正确的运行一个或多个进程。但是，细心的同学们可能会注意到，我们每个进程的地址空间需要事先通过编译确定，这样多个进程才能同时在一个地址空间内运行而不互相干扰。

但是，如何事先就预知有哪些进程会在一起运行呢？我们能不能动态的加载一个进程到任意一个空闲的地址区域内呢？要做到这一点，就需要用到虚拟内存机制。虚拟内存，可以做到给每个进程一个“虚拟”的地址空间，每个进程“看到”的地址空间都是一样的，不需要再事先确定加载地址。

此外对于操作系统来说，安全也是一个重要的功能。系统的数据安全必须要靠数据隔离来实现，而基于虚存的内存管理，也是操作系统中用来确保数据隔离的重要机制。

在本实验中，我们将学习操作系统的虚拟内存管理机制，包括虚实地址空间的管理，换页机制等。请同学们认真思考各部分的设计，考虑操作系统的安全性和性能，完成好虚存管理的功能。

本次实验的各个任务如下，做 S-core 的同学需要完成任务一，A-core 的同学需要完成任务一至任务四，C-core 的同学需要完成所有任务：

任务一 启用虚拟内存机制进入内核，实现内存隔离机制，从 SD 卡中加载用户程序，使用户进程可以使用虚拟地址访问内存。

任务二 实现缺页处理程序，发生缺页中断时自动分配物理页面。并验证之前的进程锁实现在虚存开启的情况下依然有效。

任务三 实现换页机制，在物理内存不够时或者当物理页框不在内存中时，将数据与 SD 卡之间进行交换，从而支持将虚拟地址空间进一步扩大。

任务四 实现虚存下的多线程管理，使得一个进程可以用多个线程分别执行不同的任务。

任务五 实现共享内存机制，使得两个进程可以使用共享的一块物理内存，并用共享内存机制完成进程通信。

任务六 实现 copy-on-write 策略，并编写带有 fork 功能的测试程序进行验证。

各个 core 的同学需要完成的任务如下表所示，另外 C-core 的同学需要全程使用双核，A-core 和 S-core 的同学不需要。

评分等级	需要完成的任务
S-core	启用虚拟内存机制进入内核，实现虚存下的用户进程启动和静态页表
A-core	实现多线程创建和管理，实现缺页中断处理、按需调页和换页机制
C-core	实现共享内存机制，实现 copy-on-write 策略

表 P4-1: 各个等级需要完成的任务列表

2 本章解读

这一部分的要点是：

1. 理解 RISC-V 处理器的虚存机制
2. 理解页表的基本原理和实现

本章最重要的就是**理解地址是怎么映射的**。说简单点，其实虚存机制就是一套虚拟地址到物理地址的映射。快表（TLB）相当于页表的高速缓存。那么页表是什么样的结构呢？一说到映射，很多人想的可能是一个页表项需要存储一个虚地址一个物理地址，然后每次查整张表匹配虚地址，然后再找到对应的物理地址。但实际上，**页表只存物理地址。你可以把页表想想成一个数组，数组的下标是虚地址，数组的内容是下标所对应的物理地址**。当然，虚地址空间很大，而且可能很多我们都不会用到，所以可以采用多级页表，每次索引虚地址的几位，查到下一级页表的物理地址，直到查到最后一级页表，页表项里面存的才是对应的物理地址。

最后，再次强调一下本章的**核心要点：页表存储的都是物理地址，处理器访问的都是虚地址**。

3 虚存机制的开启

说到内存，同学们想必已经不再陌生，不仅是因为每台计算机中都有内存，而且同学们在之前的 Project 中，已经知道了我们操作系统的 bootblock 会放在 0x50200000 的位置，并且能将用户程序加载到对应的内存地址并执行。不仅如此，同学们在调试中可能也会偶尔发生系统报告 page fault 的错误，报这样的错是因为访问到了 0x50000000-0x60000000 之外的地址。

内存中保存了程序所需的所有代码和数据，其内容不能被随意篡改，也不应该被其他的程序随意访问。因此，安全性是操作系统最重要的功能之一，在进行操作系统设计时必须予以考虑。通过理论课的学习，我们已经了解到，操作系统通过虚拟内存的机制来实现对内存数据的保护，但是我们研讨课所编写的操作系统到目前为止显然还没有这样的功能，所以在本实验的第一个任务中，我们就先把操作系统最基本的虚存机制建立起来。

3.1 虚存的基本概念

虽然大家理论课已经学过，但这里为了便于理解还是简述一下虚存的概念。大家可以回忆一下前面我们的用户测试程序的加载，内存在我们看来就好像一个巨大的连续的数组，我们的每个任务都用了其中的一个部分。这就造成了很麻烦的问题，即如果有某个程序不小心访问或者修改到了其他程序的内存，会导致一些莫名奇妙的问题甚至严重的数据安全性问题。

为了解决这个问题，人们设计了虚存机制。回忆一下，**我们前面实现的进程调度，让每个进程都认为自己是在独享 CPU**。但实际上，每个进程是在分享 CPU 的处理时间的。虚存也是类似，**我们希望让每个进程都认为自己在独享整个内存，但实际上是在分享物理内存**。

让进程分享处理器的执行时间的方法是，**将处理器的处理时间划分为时间片，每个程序享有一部分时间片**。分享内存的方式也很类似，我们**将物理内存切分为固定大小的页框 (page frame)，每个进程分享一定数量的物理页框**。在进程管理中，为了让每个程序都认为自己独占了处理器，我们在进程切换时对进程的上下文进行了保存和恢复。内存管理也是一样，我们需要为让每个进程都认为自己独占了内存，所以，进程访问的地址并不是真正的物理地址，而是我们为它虚拟出来的地址（这也是为什么我们说程序访问的都是虚地址）。我们为每个进程虚拟一个独立的地址空间，进程访问的地址都是这个空间里的地址。虚地址空间也按照相同的页面大小划分，然后设置好哪个虚页对应哪个物理页框。这样当程序访问一个虚地址的时候，我们就可以把它换算成对应的物理地址（这种换算一般由处理器自动完成），从而实现让多个程序分享物理内存。而**记录虚页和物理页框对应关系的数据结构就叫页表**。

3.2 内存控制相关的 CSR 寄存器

MMU（Memory Management Unit）是一种负责处理中央处理器（CPU）的内存访问请求的计算机硬件。它的功能包括虚拟地址到物理地址的转换（即虚拟内存管理）、内存保护、中央处理器高速缓存的控制。在这里我们主要关心虚拟地址到物理地址的转换功能。

默认情况下 MMU 未被使能（启用），在这时 CPU 的所有访存地址都作为一个物理地址交给对应的内存控制单元来直接访问物理内存。而 S 特权级的 SATP 寄存器则可以用来启用分页模式。启用分页机制后在 S 和 U 特权级下的访存都被视为虚拟地址，需要经过 MMU 转换为一个物理地址来进行真正的访存。

MMU 在每次进行虚实地址转换时，都会根据 SATP 寄存器中记录的物理地址去查找页目录，并根据当前访问的虚地址检索对应的页表项，得到实地址，完成地址翻译的过程。SATP 寄存器的结构如图P4-1所示。

- PPN 代表页目录自身所在位置的物理页框号，页目录的起始地址必须按照 4KB 对齐。
- ASID 表示当前地址空间的 id，这个是为了区分不同的进程，每个进程都有自己独立的地址空间，ASID 标识的就是进程的地址空间的 id。

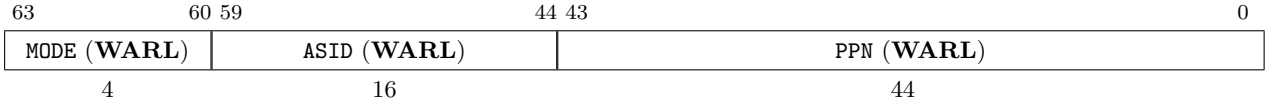

图 P4-1: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.[1]

- MODE 部分表示当前的地址翻译的模式，其编码格式如表P4-2所示。

RV64		
Value	Name	Description
0	Bare	No translation or protection.
1–7	—	*Reserved*
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.
10	*Sv57*	*Reserved for page-based 57-bit virtual addressing.*
11	*Sv64*	*Reserved for page-based 64-bit virtual addressing.*
12–15	—	*Reserved*

表 P4-2: Encoding of `satp` MODE field.[1]

代码框架中的 arch/riscv/include/pgtable.h 头文件包含了 Sv39 虚存需要用到的宏定义，其中的 set_satp 函数可以用于设置 SATP 寄存器。该函数有三个参数，分别对应着 SATP 的 MODE，ASID 以及 PPN。例如如下代码将打开 Sv39 虚存模式并将页目录设置为 0x51000000 处的物理页框。

```
// SATP_MODE_SV39 为 8, ASID 为 0, NOMAL_PAGE_SIZE 为 12, 代表 4KB 的偏移
// 这些宏定义定义在 arch/riscv/include/pgtable.h
set_satp(SATP_MODE_SV39, 0, 0x51000000 >> NORMAL_PAGE_SHIFT);
```

3.3 页框（page frame）

在理论课上我们了解到，虚存机制的核心是分页机制。如图P4-4所示，在分页机制中，虚拟地址空间和物理地址空间都被划分为固定大小的页框，而且虚拟地址和物理地址之间的映射也是通过页之间的映射来实现的。在本任务的第一步，我们就要首先将物理地址空间划分为一个个页框，用于后续的分页机制构建。

3.4 地址格式

我们使用的是 64 位 RISC-V 处理器，因此需要支持超过 32 位的虚地址。RISC-V 一共支持三种虚存模式：Sv32、Sv39 和 Sv48。这三种模式的区别主要在于支持的虚地址空间的大小不同。Sv32 支持 32 位虚地址，Sv39 支持 39 位虚地址，Sv48 支持 48 位虚地址。分别需要采用二级、三级、四级页表。

需要注意的是在 64 位架构下的确虚地址应该是 64 位的位宽，但是在启用 SV39 分页模式下，只有低 39 位有意义。SV39 分页模式规定 64 位虚拟地址的 [63:39] 这 25 位必须和第 38 位相同，否则 MMU 会直接认定它是一个不合法的虚拟地址。通过这个检查之后 MMU 再取出低 39 位尝试将其转化为一个 56 位的物理地址。

由于我们的物理内存只有 256MB，实现四级页表太过麻烦且没有必要。因此，我们选择 Sv39 模式，采用三级页表进行索引。Sv39 支持的虚地址和物理地址格式如图P4-2、图P4-3所示 [1]。在分页管理的模式下，单个页面的的大小设置为 4KB（在后面我们还会涉及到大页的概念，请大家注意区分），每个虚拟页面和物理页帧都对齐到这个页面大小。因此虚拟地址和物理地址都被分成两部分：低 12 位，即 [11:0] 被称为页内偏移 (Page Offset)，它描述一个地址指向的字节在它所在页面中的相对位置。而虚拟地址的高 27 位，即 [38:12] 为它的虚拟页号 VPN，同理物理地址的高 44 位，即 [55:12] 为它的物理页号 PPN，页号可以用来定位一个虚拟/物理地址属于哪一个虚拟页面/物理页帧。

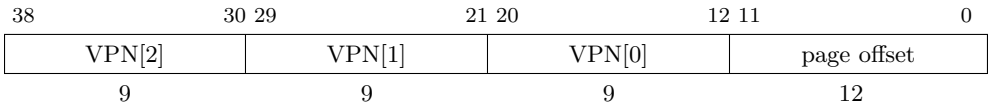

图 P4-2: Sv39 虚地址

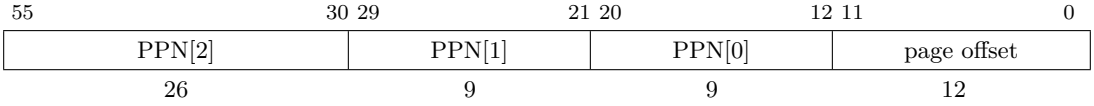

图 P4-3: Sv39 物理地址

因此我们可以将地址翻译的过程简单的理解为 MMU 拿到 39 位的虚地址后。取出前 27 位虚拟页号 (VPN)，根据该虚拟页号去页表中查找对应 44 位的物理页号 (PPN)，如果成功的查找到了，则将 44 位的物理页号与虚拟地址的 12 位偏移前后拼接作为查找到的物理地址。这样就完成了虚拟地址到物理地址的查找过程。

页框是管理物理内存的基本单元，因此页框的大小决定了物理内存分配的粒度。在现有的经典计算机系统中，大部分的页面被划分为 4KB 大小，同时搭配一些更大的页面混合使用。

3.5 Sv39 页表项说明

在物理地址和虚拟地址空间都已经被划分为页框之后，页表就用来保存从虚拟页到物理页的映射。**需要注意的是，页表本身也需要占用内存的一块空间（例如多个物理页框）**，因此在本任务中，我们需要在初始化的时候在内存中划分一块地址空间，用来存放页表。页表中的每一项称为页表项（page table entry, PTE），它的抽象数据结构如图P4-5所示，它们保存了虚拟地址到物理地址的映射关系。

如上图所示，页表项中不仅包含了物理页号（[53:10] 位），还有许多标志位（[7:0] 位），他们标志着页面的 8 个不同属性。具体说明如下：

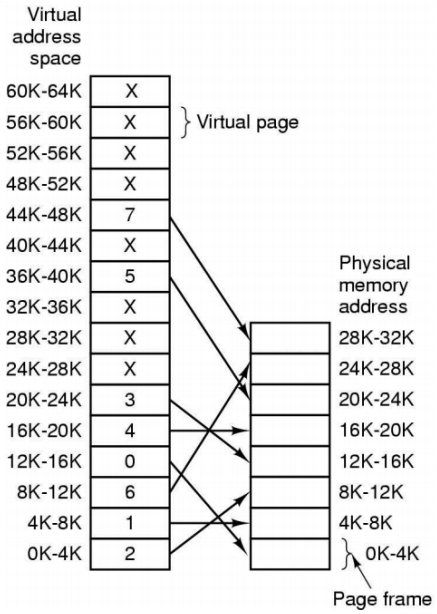

图 P4-4: 分页机制和页框

63 54	53 28	27 19	18 10	9 8	7	6	5	4	3	2	1	0
Reserved	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
10	26	9	9	2	1	1	1	1	1	1	1	1

图 P4-5: Sv39 页表项

- R、W、X 代表可读可写可执行。
- G 代表是否全局映射，对于全局映射的页，访问的时候会忽略 ASID，这是为了避免一些全局页表项被反复放入到 TLB 中。在本实验中可以暂时不予理会。
- A 和 D 代表 Access 和 Dirty，用于表明页面是否被访问或者写入过。在此处需要特别注意，**这两位的具体行为取决于硬件的实现**。有两种可能，一种是由硬件控制，当一个页被访问或者被写入的时候，硬件会自动把 A 或 D 位置位。另一种是硬件会直接产生缺页异常。我们的板子上的 RISC-V 核是后一种实现。如果 A 为 0 且发生了对这个页面的访问，或者 D 为 0 且发生了对这个页面的写入，都会直接触发缺页异常。这里需要注意的是，**由于写入也算访问，所以实际上在写入时 A 位与 D 位均应被置位**。
- U 代表 User 位，当 U 为 0 时，该页面在 User-mode 下访问会触发缺页异常；当 U 为 1 时，该页面仅在 User-mode 下可访问。**如果 SSTATUS 寄存器的 SUM 位置为 1，则 Supervisor 态下也可访问 U 位为 1 的页面**。
- V 代表 Valid 位，当 V 被置位时，该页表项有效。

当系统出现缺页异常时，需要同学们考虑上述 bit 位未设置正确的情况。

3.6 页表与地址转换过程

多级页表

一般而言，页表的设计都遵循一个页表刚好占一页这样的模式。一页按照 4KB 计算，RISC-V 一个页表项需要 8 Byte, 因此，一个页表可以容纳 512 个页表项。512 相当于 2 的 9 次方，因此，VPN 都设计为 9 位。这样便可将 VPN 作为下标定位到节点中的页表项。

这里稍微说明一下叶节点（末级页表）与非叶节点（页目录表，非末级页表）的概念，还记得前面页表项中提及到的标志位吗？其中的 X/W/R 位的不同组合有着不同的含义，具体如表P4-3所示:

X	W	R	Meaning
0	0	0	Pointer to next level of page table.
0	0	1	Read-only page.
0	1	0	Reserved for future use.
0	1	1	Read-write page.
1	0	0	Execute-only page.
1	0	1	Read-execute page.
1	1	0	Reserved for future use.
1	1	1	Read-write-execute page.

表 P4-3: Encoding of PTE R/W/X fields.[1]

可以看到，倘若 R/W/X 位均为 0，则标志着该节点为非叶节点，否则为叶节点。对于叶节点而言，需要保存 512 个页表项，该页表项中的物理页号与虚地址的页内偏移进行拼接即可得到最终的物理地址。对于非叶节点来说从功能上它只需要保存 512 个指向下级节点的指针即可，不过我们就像叶节点那样也保存 512 个页表项，这样所有的节点都可以被放在一个物理页帧内，它们的位置可以用一个物理页号来代替。当想从一个非叶节点向下走时，只需找到对应的页表项的物理页号字段，它指向了下一级节点的物理页帧，这样非叶节点中转的功能也就实现了。

地址转换过程

Sv39 地址转换的过程如图P4-6所示。这里我们展示三级页表的转换过程，这个过程是由硬件自动完成的，软件需要完成的工作是设置好一个新的进程的三级页表项并设置 satp 寄存器。假设我们有虚拟地址 {VPN2[38:30], VPN1[29:21], VPN0[20:12], offset[11:0]}:

- 操作系统记录装载「当前所用的三级页表的物理页」的页号到 satp 寄存器中;
- VPN2 作为偏移在第三级页表的物理页中找到第二级页表的物理页号;
- VPN1 作为偏移在第二级页表的物理页中找到第一级页表的物理页号;

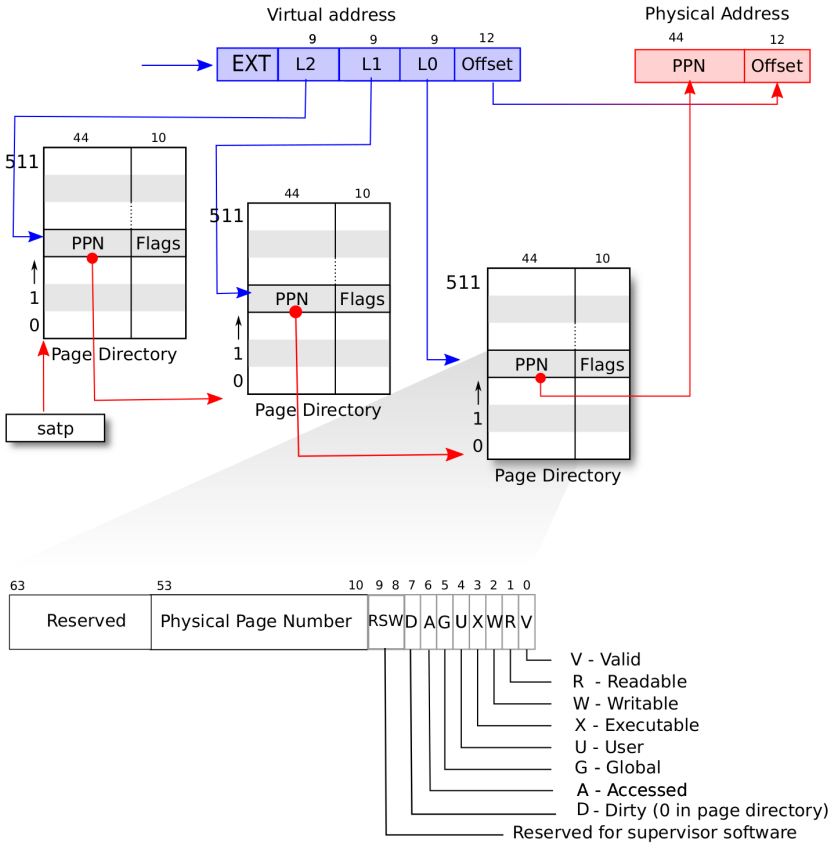

图 P4-6: Sv39 虚实地址转换过程 [2]

- VPN0 作为偏移在第一级页表的物理页中找到要访问位置的物理页号；
- 物理页号对应的物理页基址（即物理页号左移 12 位）加上 offset 就是虚拟地址对应的物理地址。

图中的页表查找过程可以用 C 语言伪代码表示为：

```
PTE* pgdir[512]; // 假设这是最高一级的页目录
if (pgdir[vpn2] != 0) { //假如对应的页表项不是空的
    // 从页目录项中获取下一级页目录的虚地址
    // 这里注意，页目录项中存的是物理地址
    // 但 CPU 都是通过虚地址访问的，因此这里需要转换一下
    PTE* second_level_pgdir = get_va(pgdir[vpn2]);
    if (second_level_pgdir[vpn1] != 0) {
        PTE* last_level_pgdir =
            get_va(second_level_pgdir[vpn1]);
        if (last_level_pgdir[vpn0] != 0) {
            uint64_t ppn =
                get_ppn(last_level_pgdir[vpn0]);
            // 这里查找到的 ppn 就是物理页号
            // 加上页内偏移就可以形成物理地址
        }
    }
}
```

这里额外提示一个问题，很多操作系统书上会讲，多级页表可以省空间。很多人在这里会有所困惑。以二级页表为例。假如有 32 位的地址空间，每个页表项 4Byte，每页

4KB。如果只用一级页表，那么管理全部 32 位地址空间需要 4GB/4KB*4Byte=**4MB** 的空间来存放页表。对于二级页表来说，需要一个页目录，页目录为 4KB，1024 项（负责索引高 10 位）。每一相对应一个二级页表，二级页表也是 4KB，1024 项（负责索引接下来的 10 位）。一共需要页目录 4KB+1024*4KB（二级页表）=**4MB+4KB**。看到这里，你会觉得很奇怪。**为什么二级页表反而需要更多的开销？**这是因为，二级页表节约空间是基于这样一个观察：**地址空间中很多的页都是用不到的**。例如一个很小的程序，它可能只需要 4KB 作为栈空间，外加 4KB 存储代码段和数据段，就足够运行了。那么地址空间中其他的页它都没有用到，有必要为它分配那么多的页表来管理吗？所以，如果有多级页表，那么只有页目录是必须的，下面的几级页表都是用到了才会分配的。对于程序没有使用的虚地址空间，系统根本不会浪费内存去记录它。这才是多级页表节约空间的根本原因。

就像上面的伪代码展示的，多级页表的查找过程中，每一级都会判断对应的页表存不存在。不存在就触发 page fault 异常，等待操作系统处理，存在则继续进行地址转换的动作。当然，我们在 S-core 的要求中并不要求大家完成页表的动态分配，为了简化，大家可以在一开始的阶段把所有页表都静态填好。

3.7 快表（TLB）及 TLB 相关指令

MMU 中的快表（TLB, Translation Lookaside Buffer）作为虚拟页号到物理页号的映射的页表缓存。充分利用计算机中的局部性原理，避免多级页表访问时的多次访存操作，可以大大提高地址转换的效率。TLB 本身在正常访问过程中是由硬件自动填充和替换的，我们作为操作系统开发者并不需要在这种情况下去操作 TLB 内容。但是有一种情况需要我们用软件来操作，就是在 TLB 中的内容已经失效，需要清空时，我们需要使用 sfence.vma 命令来清空整个 TLB。具体来讲，进程切换时需要同时进行 SATP 寄存器的切换，但如果切换了 SATP 寄存器，说明此时快表里面存储的映射已经失效了，**这种情况下内核要在修改 SATP 的指令后面马上使用 sfence.vma 指令刷新清空整个 TLB；同样，我们手动修改一个页表项之后，也修改了映射，但 TLB 并不会自动刷新清空，我们也需要使用 sfence.vma 指令刷新 TLB**。sfence.vma 可以跟上一个虚拟地址来刷新指定的虚拟地址在 TLB 中对应的页表项，当不跟任何地址时默认全部刷新 TLB。这个指令我们已经内联到 SATP 寄存器的设置函数中，并且在 arch/riscv/include/pgtable.h 头文件中封装好了基于它的 local_flush_tlb 系列内联函数，可供使用。

3.8 虚存的软硬件协同

读到这里，不知道同学们是否已经基本清楚了要在系统中实现虚存机制要做哪些事情，如果还觉得有不清楚的地方，希望大家能复习前面的内容或者查阅相关的资料，在确认好概念的理解之后再开始真正动手编写代码。这一节我们也会分别整理软件和硬件都做哪些事情，方便大家的理解：

硬件的工作

虚拟内存中硬件的工作可以总结如下：

- 判断当前访存模式是否为虚拟内存访问，如果是则通过快表（TLB）或者从 SATP 寄存器中记录的物理页号查找页目录检索对应页表项，得到实地址。
- 自动进行虚拟地址到物理地址的转换，硬件会自动根据当前的虚拟地址进行多级页表的查找工作。根据当前节点是否为叶节点来进行物理页号和地址偏移拼接得到最终物理地址或者继续检索下一级页表。
- 进行权限检查，根据检索到的页表项中的 X/W/R 位进行权限判断，如果权限错误则触发对应例外。
- 检查虚拟地址是否存在对应的映射即是否缺页，如果缺页在则触发对应的例外。

上述提到的例外在 Project2 任务书中的例外表有相关的说明，请大家注意查看。

软件的工作

其实虚拟内存中软件的工作也就是操作系统的工作，可以总结如下：

- 操作系统可以控制当前的访存模式，即通过管理相应的 CSR 寄存器来确定访存模式。
- 操作系统需要完成页表目录的建立并将根页表的物理页号填入 SATP 寄存器中，在进程切换时也需要对应的切换 SATP 中的物理页号。
- 当发生访存例外时操作系统需要根据例外的种类并进行处理。
- 快表（TLB）的刷新。

看到这里，我们已经基本了解了虚存机制的实现方式，我们下面将进入各个任务，首先给内核打开虚存机制，然后由内核给用户程序打开虚存机制，再来逐步完善我们的虚存机制。

3.9 任务一：启用虚存机制进入内核

内核虚地址空间介绍

为了能使得我们的内核能够在虚拟地址上运行，我们在编译时就需要将内核的地址空间安排为虚拟地址空间即将内核的入口设置为内核虚拟地址。在启用 SV39 分页模式下，39 位虚地址最高位为 1 意味着该空间为内核空间，否则为用户空间，即用户与内核平分 512GB 的虚拟内存空间。在这里我们严格遵循这个规范。

在内核页表初始化时，我们直接将物理地址空间与带有 0xffffffc0 前缀的虚地址空间（即内核虚地址空间）进行一一映射，开启虚存后，内核就运行在这个虚地址空间中，与运行在物理地址空间相比，只是添加了一个地址前缀。我们在 Makefile 中将内核的入口设置为 0xffffffc050202000，也就意味着内核实际被搬运到物理地址 0x50202000。

这里有一个小技巧：为了内核管理物理内存方便，一般内核虚地址和物理地址之间都是线性映射的。例如假设物理地址是 0x50200000，它会被直接映射到 0xffffffc050200000。

如果物理地址是 0x55200000，它会被直接映射到 0xffffffc055200000。这样做是为了管理页面方便，比如我们如果想在内核中访问某个进程的页表，我们先根据页目录，找到下一级页表的物理地址，然后直接加上这个固定的偏移量，就能得到相应的内核虚地址。通过这个内核虚地址，我们就可以访问到下一级的页表；如果我们想要为应用程序分配一个 4KB 的物理地址页面比如 0x55201000，其实也就是将内核地址空间中的 0xffffffc055201000 处的页面分配出去。通过这样的方式，内核便可轻易的管理整个物理内存。(注意，我们的程序访问东西只能通过虚地址进行，没法直接用物理地址访问，所以我们必须知道页表的物理地址对应的内核虚地址)。

打开虚存机制进入内核

当 CPU 上电初始化时，虚存机制并没有打开，并且也没有初始化好的页表，代码运行在物理地址上。因此内核需要一小段运行在实地址上的代码来设置内核页表并开启虚存。因为这段代码与内核编译到一起，里面的地址都已经在编译时按照内核虚地址计算好，因此它必须做到地址无关。即要求所有的跳转和数据寻址都不能使用绝对地址，只能相对于当前 PC 计算。这对代码的编写提出了严格的要求，比如不能使用 switch case 语句；不能使用指针数组等等。我们将内核的入口指定为这一段代码，当 bootblock 搬运内核到对应的物理地址后立即跳转到内核入口开始执行这一段代码。

这一段代码我们已经在 start_code 框架中提供，核心代码位于两个代码文件中，分别是 arch/riscv/kernel/start.S 和 arch/riscv/kernel/boot.c。

在编译时将内核的入口指定为 _boot，这段代码位于 arch/riscv/kernel/start.S 中。

arch/riscv/kernel/boot.c 中的 boot_kernel 函数将会建立内核页表，把内核空间映射为 2MB 的大页跳转到内核的真正入口。核心代码如下：

```
extern uintptr_t _start[];
/********** start here **************/
int boot_kernel(unsigned long mhartid)
{
  if (mhartid == 0) {
    setup_vm();
  } else {
    enable_vm();
  }

  /* 进入内核后将永远不会返回 */
  ((kernel_entry_t)pa2kva(_start))(mhartid);

  return 0;
}
```

boot_kernel 的参数为核 id mhartid。需要注意的是主核和从核各有一个 SATP 寄存器。显而易见，主核需要完成内核页表的映射并设置 SATP 寄存器打开虚存，而从核只需要设置好 SATP 寄存器即可。当打开虚拟内存机制后，需要跳转到 _start 即之前实验中的内核入口，需要注意的是因为此处是相对于当前 pc 计算得到的 _start 地址，因此，需要将该地址转化为内核虚地址才能进行跳转。

目前整个内核的结构变成了 _boot->boot_kernel->_start->main。前两个阶段运行在实地址上，只访问物理地址。随后都运行在内核虚拟地址上，只访问虚地址。

实验要求

启动虚存机制进入内核。

实验步骤

1. 请大家仔细阅读注意事项后再开始本次任务。
2. 修改 arch/riscv/boot/bootblock.S，将内核搬运到 0x50202000 处。
3. 内核页表映射的代码我们已经为大家写好，请大家认真阅读并理解代码实现。熟悉这一部分的代码对后续建立用户页表有所帮助。
4. 完成 arch/riscv/include/pgtable.h 中的 API。

注意事项

1. 清空内核 BSS 段的操作不能由 _boot 完成，因为此时还运行在实地址空间。
2. boot_kernel 的运行也是需要一个栈的，这个栈当然是实地址栈。start_code 将 0x52001000 到 0x52002000 这一段地址作为 boot_kernel 的栈。
3. 因为内核编译时已经按照虚地址编译，因此导入符号表时也是虚地址。可以在 make gdb 后在 gdb 中输入命令 add-symbol-file build/main 0x50202000 动态加载内核的符号表到地址 0x50202000。这样便能使用 gdb 调试内核页表映射部分的实地址代码了。
4. 建立完物理地址到内核虚地址的映射以后，就可以开启虚存机制，然后加载内核了。这里需要注意的是，因为 boot_kernel 部分的代码是运行在物理地址上的，开启了虚地址以后，boot_kernel 的代码就也会通过虚存机制来访问内存。为了让 boot_kernel 的代码不出问题，所以需要临时把 0x50200000 到 0x51000000 所在的内核代码空间映射到 0x50200000 到 0x51000000 上，这样才能让 boot_kernel 的代码在虚存开启的情况下也能正确运行。这一映射方式已经由 start-code 提供好，在进入内核后，**同学们需要将这一临时映射取消掉**，避免后面用户程序用到这部分虚拟地址空间与内核地址冲突。
5. 我们板子上的 bios_sdread 没法一次读取太多的 sector。建议按照 64 个 sector 一组分多次读入。
6. 内核进入 _start 并完成 BSS 段的清空后，便可继续开始下面的任务。

3.10 地址空间管理

在 Project2/3 中，我们在开发版和 QEMU 上利用 DASICS 功能来实现对实地址空间中用户地址区间与内核地址区间的隔离。在打开虚存之后内核与用户空间，以及进程之间都有了比较好的内存隔离。因此在本实验中，我们取消 DASICS 检查的机制，在

QEMU 和板子上使用 loadboot 和 loadbootm 直接启动双核或者多核，而不需要再使用 loadbootd。

在 project4 及其之后的实验中，**内核将管理所有的物理地址**。前一节中我们已经提到内核虚地址和物理地址是线性映射的关系，因此在本实验中我们以内核的视角来管理和分配地址空间。建议的地址空间划分如所表P4-4所示。

地址范围	建议用途
0xffffffc050000000-0xffffffc050200000	BBL 代码及其运行所需的内存
0xffffffc050200000-0xffffffc051000000	Kernel 的数据段/代码段等
0xffffffc051000000-0xffffffc052000000	Kernel 页表以及跳转表等
0xffffffc052000000-0xffffffc060000000	供内核动态分配使用的内核虚拟地址空间

表 P4-4: 地址空间用途划分

4 用户进程

之前的实验中，我们用的用户进程都更接近于线程的感觉，因为各个进程之间的地址空间并没有真正隔离开，他们的运行空间都是事先约定好的。这一次，每个应用程序的入口地址都将是一样的，不需要严格的限制，并在操作系统中加载执行，拥有各自的虚地址空间。

4.1 shell

本次实验中，我们会继续使用之前的 shell。本次需要支持 4 个命令，即 exec、kill、ps、clear，其要求与 P3 中相同。

4.2 加载用户程序

本次实验中，bios 提供了 sd 卡读写的相关函数，**注意传入的地址参数为物理地址**。**在进入到内核的 _start 后，所有地址访问都使用虚地址，内核当中不要出现任何的直接对物理地址的访问**。

为用户程序建立页表映射并拷贝数据到映射的物理页面

在打开虚存之后，用户程序的加载就不再是把用户程序从 SD 卡中读到内存中的对应地址并执行那么简单了。我们需要在加载用户程序时就建立好用户程序的页表和虚实地址映射，这样才能在切换到用户进程后能运行在对应的虚拟地址空间上。在之前的实验中大家已经知道，内核启动后会首先加载用户进程 shell，随后在 shell 中使用 sys_exec 系统调用启动新的用户测试程序。在这两种方式中都存在一个共同点：**在当前进程的地址空间加载另一个进程**，我们可以这样理解：子进程的地址空间映射是由父进程完成的，因此这是一个跨地址空间的操作。对于三级页表而言，为子进程的某个虚拟页面

分配好物理页面并建立好页表映射之后，还需要载入用户程序到对应的虚地址，即将内容拷贝到映射的物理页面中。，因为 CPU 最终的访存其实使用的还是物理地址，接下来我们将说明如何实现这一操作。

start_code 中的 include/os/mm.h 头文件中中定义了一个名为 alloc_page_helper 的函数。该函数为指定的虚地址建立页表映射，他的定义如下：

```
// va 为需要映射的虚拟地址，pgdir 为页表目录，
// 返回值为为 va 映射的物理地址对应的内核虚地址
uintptr_t alloc_page_helper(uintptr_t va, uintptr_t pgdir);
```

alloc_page_helper 为指定的虚拟地址 va 建立好页表映射之后将返回为 va 映射的物理页面对应的内核虚地址，我们只需要将 SD 卡中的内容加载到该内核虚地址上即可。

用户程序页表映射细节

在 Project1 中，大家已经实现了将测试程序的 elf 文件制作成为镜像。此处我们对 elf 文件做进一步的探讨，首先给出 64 位的 elf 文件程序头表数据结构如下：

```
/* Program segment header.  */

typedef struct
{
  Elf64_Word    p_type;                 /* Segment type */
  Elf64_Word    p_flags;                /* Segment flags */
  Elf64_Off     p_offset;               /* Segment file offset */
  Elf64_Addr    p_vaddr;                /* Segment virtual address */
  Elf64_Addr    p_paddr;                /* Segment physical address */
  Elf64_Xword   p_filesz;               /* Segment size in file */
  Elf64_Xword   p_memsz;                /* Segment size in memory */
  Elf64_Xword   p_align;                /* Segment alignment */
} Elf64_Phdr;
```

我们使用 linux 的 readelf 工具获取一个简单的测试程序的程序头表信息，得到如下结果：

```
  Elf file type is EXEC (Executable file)
  Entry point 0x10000
  There are 2 program headers, starting at offset 64

  Program Headers:
    Type           Offset             VirtAddr           PhysAddr
                   FileSiz            MemSiz              Flags  Align
    LOAD           0x0000000000001000 0x0000000000010000 0x0000000000010000
                   0x000000000000c35 0x0000000000001c40  RWE    0x1000
    GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                   0x0000000000000000 0x0000000000000000  RW     0x10

   Section to Segment mapping:
    Segment Sections...
     00     .text .rodata .sdata .sbss .bss
     01
```

该结果中程序头表的 Type 到 Align 字段分别对应 ELF64_Phdr 结构体中的 p_type 到 p_align 字段。在加载可执行文件时，我们只需要关心 p_type 为 LOAD 类型

的字段。这里需要提醒一下，为了尽可能的简单，在我们的实验框架中，只会存在一个 LOAD 类型的程序段。罗列出我们实验中需要关心的成员及其含义如表P4-5所示。

成员	含义
p_vaddr	“Segment” 第一个字节在进程虚拟地址空间的起始地址，在我们的实验中，**这个虚拟地址等于进程的入口**。
p_filesz	“Segment” 在 ELF 文件中所占空间的长度，可能为 0，因为有可能这个 “Segment” 在 ELF 文件中不存在内容。
p_memsz	“Segment” 在进程虚地址空间中所占的长度，它的值也可能为 0。
p_flags	“Segment” 的权限属性，比如可读 “R”，可写 “W” 和可执行 “X”。

表 P4-5: 程序头表成员及其含义

程序头表内的 p_filesz 表示当前 segment 在 ELF 文件中所占的大小，p_memsz 表示当前 segment 被搬运并展开到内存中所占用的大小。一个 segment 是由若干个 section 组成的，而对于其内数据都为 0 的 bss 段而言，没必要在浪费宝贵的文件空间来保存一段全为 0 的数据，因此 ELF 文件的设计者想出了一个聪明的办法：只要让 p_memsz 大于等于 p_filesz，多余的部分由操作系统在装载 ELF 文件的时候自动填 0 就行了。这样做既节省了文件存储空间，也满足了 bss 段内起始数据为 0 的假设。

因此，建立用户页表的过程即为映射从程序入口开始的 p_memsz 大小的空间，以上面的测试程序为例，我们需要为应用程序映射 0x10000 ∼ 0x11c40 的虚拟地址空间即两个 4KB 的页面。随后需要将 SD 卡中长度为 0xc35 的数据拷贝到为虚拟地址 0x10000 ∼ 0x10c35 映射的物理页面中，即假设为 0x10000 ∼ 0x10c35 映射了 0x52000000 ∼ 0x52000c35 的物理地址空间，则需要将数据拷贝到 0xffffffc052000000 ∼ 0xffffffc052000c35 的内核虚拟地址空间。并将为 0x10c35 ∼ 0x11c40 映射的物理地址空间填 0 以满足 bss 段起始数据为 0 的假设，这一动作也可以在 crt0.S 中进入到用户程序的 main 之前完成。

初始化进程页表项标志位设置

首先，对于非叶节点的页表项，需要将 V 位置位，代表页表项有效，其余的标志位不需要置位。

对于叶节点的页表项：

- V：置位，代表该页表项有效
- R/W/X：程序头表中 p_flags 字段则对应着页表叶节点页表项的 R/W/X 位。在我们的实验中，p_type 为 LOAD 类型字段的权限都为 RWE，因此在为用户程序建立页表时，需要将末级页表项的 R/W/X 位全部置位。
- U：置位使得用户态下能够访问这些页表。
- G：无需关心。

- A/D：在 QEMU 上，当访问该页面时，会根据访问或者写入自动置位；在我们的板子上，如果 A 为 0 且发生了对这个页面的访问，或者 D 为 0 且发生了对这个页面的写入，都会直接触发缺页异常。因此我们建议大家在完成任务一和任务二时将这两位置一。A/C-core 在后续的换页机制中，如果大家设计的换页算法比较好，是需要根据这两位来选择换出的页面的，这时请大家初始化时就不要设置这两位，等到发生了相应的例外再处理。S-core 的同学在初始化时请将这两位置为 1。

需要注意的是，我们板子上的 bios_sdread 没法一次读取太多的 sector。建议按照 64 个 sector 一组分多次读入。

4.3 任务一续：执行用户程序

实验要求

加载 shell 作为第一个进程启动，支持 exec、kill、ps、clear 命令。可以执行起 fly 程序，显示小飞机。

实验步骤

1. 请大家仔细阅读注意事项后再开始本次任务。
2. 打开虚存后内核中的地址访问就都需要使用虚拟地址了，请大家将之前使用物理地址的部分进行修改。
3. 修改内存管理模块，阅读注意事项以查看更多细节。
4. A/C-core 的同学为 task_info_t 新增成员存储程序头表的 p_memsz 字段。
5. 为每个新进程分配一个 4KB 的页面作为用户页表目录并拷贝内核页表到该页面。建议大家将用户进程的页表对应的内核虚拟地址存储到 PCB 结构体中以方便管理。
6. 认真阅读理解 4.2 节的内容，修改 loader 的实现，将步骤五中的的页表目录作为参数传递给 loader，由 loader 完成用户虚实地址映射并从 SD 卡中载入用户程序，此处对于 S-core 和 A/C-core 的同学要求不同，请注意查看注意事项。
7. 修改调度器，调度时切换用户页表。
8. 修改 kill 和 exit 等系统调用的实现，回收分配的物理页。

注意事项

1. 做 S-core 的同学不需要改动 Makefile，做 A/C-core 的同学需要把 Makefile 中的 USER_ENTRYPOINT 修改为 0x10000。与 Project3 一致，做 S-core 的通过“exec [id]”的方式启动测试，A/C-core 的同学通过 “exec name” 启动测试。顺带提一下，在本章的测试中 A/C-core 的同学启动测试时加上 & 选项。

2. 我们已经将内核框架中的 KERNEL_JMPTAB_BASE 和 BIOS_FUNC_ENTRY 宏定义更改为内核虚地址，请大家注意查看，其余大家自己设计的使用物理地址的部分，请大家自行更改。

3. 内核中的 pid0_pcb 也是需要页表目录的，由于它只需要在内核空间中运行，因此它的页表即为内核页表，对应着 arch/riscv/include/pgtable.h 中的宏定义 PGDIR_PA：0x51000000lu。这是一个物理地址，请将其转化为内核虚拟地址存储到 pid0_pcb 中。

4. 本实验中我们提供了简单的内存管理模块代码以供参考，代码实现在 kernel/mm/mm.c 以及头文件 include/os/mm.h 当中，这个简单的内存管理模块中还包含了 S-core 的大页分配的简单示例。如果大家之前在 mm.c 中自己实现了内存管理算法，这两个文件的冲突可能比较多，请大家对照着 start code 仓库中的代码按需修改。

5. 对于用户页表的建立，本次实验对 S-core 和 A/C-core 有不同的要求。

   S-core：对于 S-core 而言，为了降低难度，我们要求用户进程只需要建立二级页表即 2MB 的大页。并假定用户空间只有 2MB，将其顶部作为用户栈栈顶，具体的结构如图P4-7所示。因此对于 S-core，只需要将虚地址 0x200000 映射到分配的 2MB 大页即可，页表仍需要内核分配 4KB 的页面。2MB 的大页映射可以参考内核页表的映射方法。

   这样操作的好处在于：由于目前的用户程序体量比较小，2MB 的大页存放用户进程绰绰有余，不需要过多关心 4.2 节中提到的 p_memsz 和 p_filesz 的细节；将用户程序从 SD 卡中读取到映射的物理内存中的操作会比较容易，如果是三级页表则需要以 4KB 为单位拷贝；2MB 的大页包含了用户栈空间，因此不需要单独映射用户栈。

   因为 Sv39 中页表仍然是占用 4KB 的页框，因此 S-core 需要支持分配 4KB 和 2MB 的页面，前面我们已经提到对应的页面物理地址也需要是 4KB 或者是 2MB 对齐。因此我们建议 S-core 的同学仿照 Project2/3 中的地址划分模式，划分出两块空间，分别用于分配 4KB 的页面和 2MB 的页面。

   但是操作的便利伴随着内存的浪费，并且大的页面不利于内存的细粒度管理。正如前面所介绍的，分配 2MB 的大页对于操作系统来说也是比较困难的。因此，我们鼓励大家尝试建立完整的三级页表。

   A/C-core：对于 A/C-core 而言，我们要求大家建立三级页表即 4KB 的缺省页面。并根据 task_info 中存储的信息，映射对应大小的地址空间。由于我们的应用程序占用的内存空间很有可能大于 4KB，因此可能需要分配多个页面。4KB 的页面伴随着一个问题，即两个连续的虚拟地址页面映射的物理页面不一定是连续的。因此将用户程序从 SD 卡中装载到内存中时需要以 4KB 为单位。需要注意的是 A/C-core 除了内核页表之外，其余都是三级页表，即所有的页面都是 4KB 的页面。

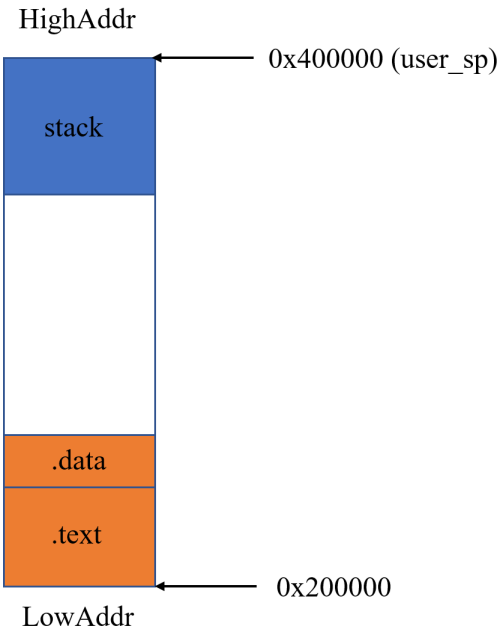

图 P4-7: S-core 用户地址空间图

6. 用户进程的内核栈的分配和以前一样，但用户栈需要设定到固定的一个位置。对 S-core 而言，该地址为 0x400000；对 A/C-core 而言，推荐设置为 0xf00010000，然后映射一个物理页给这个虚地址，作为用户栈。

   切换进程时需要把页目录和 ASID 一并切换。这里会有一个问题：页表切了，虚地址变了，后面的内核代码怎么执行？实际上，这就是我们为什么要把内核的虚地址映射到所有的页表中。内核部分的虚地址在所有进程的页目录中都是一样的，这样设计使得页目录的切换不会影响内核态的代码和数据。**这里需要特别注意的是，在将页表的基地址写入到 satp 后，需要刷新 TLB。否则可能出现 QEMU 上正确，但板子上出错的问题**。

7. 请大家注意内核初始化的时候 SSTATUS 寄存器的置位，其中 SUM 位控制着 Supervisor 态下的程序是否可以访问 U 位为 1 的页面。请大家确保 SUM 位已经被置为 1.

8. 在本实验中，对于 A/C-core 而言，Project3 中实现的命令行参数的拷贝也是一个跨地址空间的拷贝，具体怎么做到就需要大家自己来设计了。

要点解读

1. 添加了虚存机制后如何回收干净一个进程所使用的全部内存是一个大的挑战，请大家思考如何将一个进程所使用的内存回收干净。

2. 请务必记得，CPU 发出的所有访问都是虚地址，在 C 代码里想访问任何东西都要通过虚地址访问。页表中填的内容都是物理地址。

3. 为什么要将内核页表映射到用户的页表中呢? 试想，如果不将内核页表映射到用户页表中，当用户陷入内核时必然会发生缺页。所以我们需要为每个用户都映射一个内核页表。因为内核页表已经事先建立好并不会发生改变，简单来说就是在为测试进程分配页表时，将内核的页表整个拷贝过来，随后在此基础上进行用户的页表映射工作。因为在映射内核页表时已经关闭页表项的 U 位，所以用户空间不能访问到内核空间，保证了内核空间与用户空间的隔离。

5 缺页和按需调页

在任务一中，我们初始化时就将虚拟地址到物理地址的映射建立好了。但是实际系统中常常采用一种按需调页的机制，只有数据在真正被访问时，才建立虚拟地址到物理地址的映射。而这种情况下就会出现缺页：（1）软件访问的虚拟地址尚未建立虚实地址映射，或者（2）一个已建立好的虚实映射，但物理页框没有在物理内存中而是被换出到了磁盘上。

在任务二中，我们就要继续完善内存管理机制，实现按需调页，并针对上述缺页情况（1）实现对应的缺页处理程序。从这一部分开始的内容只需要 A-core 和 C-core 的同学完成。

5.1 缺页（page fault）

当读/写指令访问的地址无法在页表中找到对应的虚地址的时候，会触发读/写缺页异常，跳转到异常入口。通过识别相应的 cause 寄存器的值，可以转到对应的缺页处理程序中。触发缺页的地址会被存放在 CSR 的 stval 寄存器中。缺页处理程序建立一个从虚拟页面到物理页面的映射并将它加入页表，我们将在任务二中实现缺页处理程序。如果这个映射已经建立但是处于磁盘上时（即上述缺页情况（2)），就需要进行页替换，对于页替换的情况，我们作为任务三的内容。

5.2 按需调页（On-demand Paging）

在前面的任务中，初始化时我们就建立好了进程所需的所有页表项，建立好了虚拟地址到物理地址的映射，但是在实际系统中，一个进程所需的资源并不需要在初始化时就分配好，而是可以等到程序真正使用时再分配，这就是按需调页机制。这样的机制可以使得初始化的过程变得简单，同时可以根据程序的实际使用情况来管理内存资源，是操作系统的一种常见机制。

在本任务中，同学们需要加入按需调页。在上一个实验中，我们分配了包括加载程序的页面和用户栈空间在内的一大块空间，实际上大部分并没有被用到，这造成了空间的浪费。在这一个任务中，需要大家实现按需分配：每当程序访问一个虚地址，如果该地址没有被分配物理页面，则为其自动分配一个物理页面。这样，我们初始化程序时就只需要分配拷贝程序所需的页面了。

5.3 任务二：动态页表和按需调页

自行实现缺页处理程序，动态建立页表映射，分配物理页框。

测试用例 1

使用 fly 和 rw 两个程序作为测试用例，其中 rw 会接受命令行参数，并读写相应的地址。例如：

```
exec rw 0x10800000 0x80200000 0xa0000320
```

此时，rw 程序会访问 0x10800000、0x80200000、0xa0000320 这三个虚地址，并向其中写入一个随机数。如果所有地址均能顺利写入或读出，且写入和读出的数据相同，即判断为功能正确。地址的值在测试的时候随机输入，保证是用户态的地址（即高位是 0x000 开头的，而不是内核地址那种 0xfff 开头的）。

测试用例 2

使用 test/test_project4/lock.c 作为测试用例，功能和 P2 的 lock 测试一样，将同学们实现的进程锁在虚存开启的情况下测试功能是否依然正确。使用 “exec lock [行号] &” 在 shell 中启动多个 lock 测试。具体的现象为任何时候只有一个 lock 进程能得到锁并运行。

6 换页机制（page swap）

在完成了前两个任务之后，我们已经可以使用全部的内存空间，拥有了真正的进程，并实现了按需调页机制。而本任务则包括了换页机制（swap）和页替换算法，这些功能将使虚存的管理更加完整。

换页机制是指当系统的物理内存不够用时，将部分虚拟页面对应的内容写入磁盘的 swap 空间，在需要访问时再加载回内存的机制。换页机制使得虚拟地址空间可以大于物理地址空间，从而程序员可以不用担心物理内存的大小直接使用虚拟地址。因为当虚拟地址空间可能大于物理地址空间时，一次对虚地址的访问发生了缺页中断就有两种可能：一是这个虚拟地址第一次被访问，没有分配过物理地址；二是这个地址对应的物理页面被写入了磁盘的 swap 空间，需要从磁盘中读回到物理内存中。因此当换页机制打开时，我们需要额外的数据结构来表示，一个虚拟页面对应的数据是否被换入到了磁盘中，被换到了磁盘的哪个位置上。

6.1 任务三：换页机制和页替换算法

在本实验中，我们使用 SD 卡上的一块空间来当作系统的 swap 空间，通过读写 SD 卡的方式来进行换页操作。SD 读写使用 bios 调用实现，我们已经提供给大家。这里需要再次提醒，我们提供的 SD 卡读写调用需要大家填写的是物理地址作为参数。

注意：在换页机制执行时，除了需要设计一套页标识与管理机制外，也需要替换算法决定对哪一个物理页框进行替换。在本任务中，请同学们自行设计用于实现换页的数据

结构与管理机制，并设计一个替换算法（可以参考操作系统理论课介绍的算法，如 Clock、FIFO 等），思考物理页应该如何索引、如何替换，实现相应的替换算法。

测试用例请同学们自行思考如何有效验证。一种可行的测试用例供参考：限制能使用的物理页框个数（即物理地址空间），但实际可用的虚拟地址空间大于物理地址空间，编写一个程序对该虚址范围进行可控的访问序列，从而可以触发页替换。这个访问操作可以类似之前任务二中的 rw 程序，通过输入虚地址来控制访问序列。如果要测试一些复杂的替换算法，访问序列中需要体现对热点页面的访问。在测试时，同学们最好也通过改变访存序列，来验证替换算法是否正确实现。

回到我们的换页机制，有了 SD 卡的读写之后，我们就可以在需要分配一个物理页框但现有物理页框不够使用时，通过页替换的方式选择一个物理页框将其写回 SD 卡，从而释放该物理页框用于新的分配需求；或者虚实映射已经建立好但物理页框被换出至磁盘，通过页替换的方式将页框重新换入物理内存。当然，换回物理内存的页框不一定拥有和之前一样的物理地址。

注意事项

1. 在 QEMU 上调试本任务时，当 SD 卡读写的范围超过镜像大小时将会报错。建议在本任务中制作完成镜像后，在后方 padding 一些空间以方便 QEMU 上 SD 卡的读写。可以采用命令：

   ```
   1  dd if=/dev/zero of=image oflag=append conv=notrunc bs=512MB count=2
   ```

   该命令表示在镜像 image 后方 padding 两块大小为 512MB 的空间即 1GB，为了方便，建议大家将该命令写入到 Makefile 中。

7 多线程管理

有了虚存机制之后，我们可以完全区分开进程和线程的概念，而不用像之前那样模糊两者的区别。两个进程会拥有完全独立的虚拟地址空间，从而自然的形成数据的隔离保护。而两个线程则应该拥有完全相同的虚拟地址空间，自然的形成数据共享。只是两个线程可以执行不同的代码，可以由内核分别调度。任务四需要同学们实现线程这一机制，并完成一个多线程异步收发的 mailbox 测试。实现 mailbox 本来是 P3 中由同学们使用多进程机制完成的任务，有了线程之后，可以非常简单的利用双线程实现这样的功能。

7.1 任务四：多线程的 mailbox 收发测试

本实验任务需要同学们完成的任务是：三个进程分别收发 mail，每个进程建立两个线程分别执行发送 mail 和接收 mail 的操作。start-code 的测试程序代码为 test/test_project4/mailbox.c。

同学们需要实现的是线程相关的功能和接口，包括了测试程序中用到的 pthread_create 和 pthread_join，以保证 P3 实现的 mailbox 相关系统调用正确。pthread_create

是创建一个线程，执行指定的代码。创建完成后原线程继续执行其他的代码，而新创建出来的线程执行 pthread_create 指定的代码。这样就实现了双线程执行不同的代码。线程虽然由内核分别调度执行，但是由于共享相同的虚拟地址空间，所以都可以访问进程内的所有数据。

在 main 函数中调用的 pthread_create 函数的接口逻辑是，主线程创建一个线程描述符为 recv 的线程，它执行 recv_thread 这个函数的代码，执行 recv_thread 函数的参数为 id，作为接收线程。主线程在调用 pthread_create 创建完这个线程之后执行 send_thread。发送线程随机的给另外两个进程之一发送，发送内容为接收线程收到的字符串。即两个线程需要通过共享一个 buffer 来传递这一字符串。recv_thread 和 send_thread 里面分别调用了 sys_mbox_send，sys_mbox_recv，sys_mbox_open 等 mailbox 功能函数，这些都与之前 P3 实现的功能相同，还使用 P3 实现好的代码即可。

注意 main 函数最后还调用了 pthread_join 函数，这个函数的功能是阻塞等待 recv 这个线程执行结束之后，回收这个线程的资源。

注意事项

1. 要让操作系统能看到进程创建的线程并调度起来，pthread_create 函数势必要进行系统调用，并由内核创建好相关的数据结构，比如类似于进程控制块的线程控制块，然后加入调度队列。但是和进程区分开的是，创建线程时不需要一个单独的页表，而是使用和之前线程完全一样的页表。同一进程的两个线程之间任务切换时，也不需要切换页表。不过两个线程依然需要独立的栈空间，以独立的运行代码和控制各自的临时变量。

2. 再次请大家注意内核初始化的时候 SSTATUS 寄存器的置位，其中 SUM 位控制着 Supervisor 态下的程序是否可以访问 U 位为 1 的页面。请大家确认 SUM 位已经被置为 1，这样 mailbox 的一些需要内核把数据拷贝到用户空间的操作才不会触发例外。

3. 在 Project2 的时候做过 C-core 的同学在那个时候已经实现过一个线程的机制，请同学们思考，在有虚存和没有虚存的情况下，线程的实现有哪些区别。

8 共享内存

内存管理的最后一部分是共享内存。它将允许两个进程使用各自的虚地址访问同一块物理地址。从这里开始的部分只需要 C-core 的同学完成。

8.1 共享内存的概念和用处

共享内存可以允许两个进程将各自的虚页映射到同一个物理页面上，这样两个进程就可以简便的修改同一个内存数据。从内核的角度来说，内核需要在用户进程申请共享内存时提供这样的功能，建立好虚实映射，并管理这个物理页的回收。在有进程释放了映射到共享物理页的虚拟页时，如果有其他进程依然在使用这个物理页，则这个物理页

不能被回收。直到所有进程都释放了到这个物理页的映射，这个物理页才可以被回收。当然，同时修改共享内存的内存数据就存在操作的原子性问题，因此我们在操作共享数据的时候就需要使用原子指令进行操作。

8.2 任务五：多进程共享内存与进程通信

请大家参考测试程序完成本任务，测试程序为 test/test_project4/consensus.c。
多核共享内存需要实现两个接口：

sys_shmpageget(int key) 该接口的输入是一个 key 值，同样的 key 值会对应到同一块共享内存区域上。返回值是一个地址。这个地址是由内核寻找的一块尚未被使用的虚存地址。由内核将共享的内存映射到该地址上，并将地址返回。如果一个 key 是第一次被使用，内核会建立一个全 0 的空白物理页面作为共享页面。后续传递同样的 key 的进程得到的物理页面是同样的。从而实现共享内存。

sys_shmpagedt(void* addr) 该接口将之前用 shmpageget 获取到的虚地址解除与共享内存区域的映射。如果没有进程再使用对应的共享内存时，需要将物理页回收。

测试程序启动后会首先测试共享内存的获取与回收。在获取到共享内存的虚地址后，会首先立即解除映射，然后尝试对之前获取到的虚地址进行写入。由于映射已解除，所以写入操作会导致内核为该虚地址自动分配一个新的物理页。然后测试程序会再次尝试获取共享内存。由于之前的虚地址已经被占用，所以内核只能返回一个新的虚地址供共享内存使用。这样，同学们实现的 shmpageget 就能保证每次返回的虚拟页都是动态寻找的空闲虚拟地址，而不是每次静态指定的虚地址。

在完成上述测试后，测试程序会创建一定数量的子进程。创建子进程的过程依然使用的是 exec 接口，请同学们参见测试程序。所有子进程会获得到同一块共享内存。之后这些进程会通过共享内存和原子指令，每一轮选择一个进程号。被选中的进程会显示自己退出了（虽然并未实际退出），其他进程会显示本轮被选中的是哪个进程。之后，每一轮都会选择一个进程，直到所有的进程全部被选过一遍为止。在这一过程中，所有的进程都需要正确显示被选中的进程是哪一个。当所有进程都被选择之后，所有进程一起退出。测试程序的输出效果如图P4-8所示。

8.3 任务六：fork 和 copy-on-write 机制

copy-on-write 即写时拷贝机制是操作系统中常用的一个性能优化机制。当使用 copy-on-write 时，两个进程各自的一个虚页（假设为 A 和 B）指向了同一个物理页，并且这两个虚页都被置为只读。如果其中一个进程想要修改虚页 A，那么就会触发 page fault，操作系统会为虚页 A 分配一个新的物理页面，并把虚页 A 原本对应的物理页中数据拷贝过去。拷贝完成后再对这段数据内容进行更改，从而避免数据冲突，又降低了数据拷贝开销（因为某些页面永远不会更改，因此不需要再分配空间）。在实际操作系统中，copy-on-write 机制最能发挥作用的地方是 fork(clone) 系统调用，该调用需要拷贝进程的全部地址空间，正适合使用 copy-on-write 机制。因此本任务需要同学们自己实现一个 fork

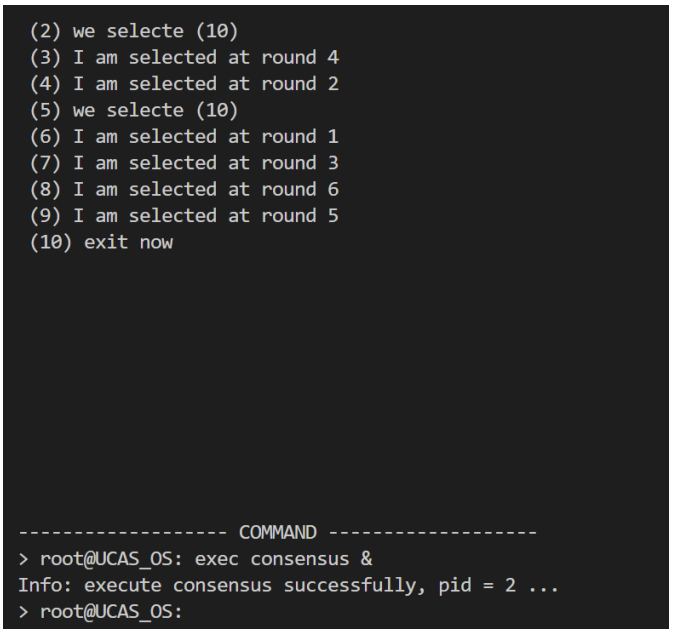

图 P4-8: 共享内存测试效果

机制，以及配套的测试程序，以正确演示 fork 的效果。这里有一个附加的要求，请同学们不要在主函数里直接调用 fork，必须先调用一个其他函数，在函数中再使用 fork 系统调用。添加这个要求的原因是，fork 之后包括堆栈的内容也需要进行复制，而堆栈中可能包含了函数的返回地址，而如果调用 fork 时堆栈内容比较简单，就难以体现出堆栈复制的重要性。值得一提的是，如果在没有虚存的条件下使用 fork，在物理地址上复制了堆栈内容之后，可能还需要根据新进程的物理地址情况，去修改堆栈里面保存的地址，但是有虚存的情况下，两个进程可以使用相同的虚拟地址，给 fork 过程中的 copy 动作带来了方便。

参考文献

[1] “The risc-v instruction set manual volume ii: Privileged architecture v1.10,” 2017.

[2] “xv6: a simple, unix-like teaching operating system,” 2012. Available at https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf.