国科大操作系统研讨课任务书

RISC-V 版本

版本 *1.0*

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。而对于操作系统来说，安全也是一个重要的功能，但是系统的数据安全必须要靠数据隔离来实现，而本实验需要实现的虚存管理，就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 实现内存隔离机制，将用户进程作为可执行文件加载并执行，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 实现虚存下的 fork 系统调用，且新创建的进程数据采用 copy-on-write 策略。

任务六 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。

任务七 对同一个并行加法任务实现多线程锁，并与多进程锁对比它们之间的效率差距。

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	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 copy-on-write 的 fork，共享内存，多线程锁

表 P4-1: 各个等级需要完成的任务列表

RISC-V 手册对于虚存的硬件工作机制有详细描述，请大家注意查看关于 SATP 寄存器的相关说明，以及 supervisor 态下的虚存机制的说明。

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**理解页表是怎么实现的**。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。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 Project 1 和 2 中，已经知道了我们操作系统的 bootblock 会放在 0x50200000 的位置。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。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。大家可以回忆一下我们前面的代码中使用内存的方式。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。第一，我们目前所有执行的任务是预先准备好的。任务中的所有的变量的地址、代码中涉及到的地址，全部是在编译期间算好的，且每个任务同一时间只能执行一个。想象一下，假如执行一个任务（含有全局变量）时，又启动了一个同样的任务，那么这两个任务使用着相同的全局变量。在我们目前设计的内核中，同一个全局变量只有一个地址。所以这两个任务肯定都会去修改这个全局变量，互相影响，导致执行错误。第二，假如有一个编写得不好的任务，不小心把别人的用户栈甚至是内核栈覆盖了，那么整个系统就直接崩溃了。

为了解决这一问题，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 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P4-1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。

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。在本实验中，同学们可以**自由选择页框的大小**，既可以使用单一的页框大小，也可以使用混合的页框大小。请采用混合页框大小设计的同学们思考，不同的页框大小对系统的性能和资源使用有什么影响？如果使用混合页框大小，应该如何进行不同大小页面的分配？

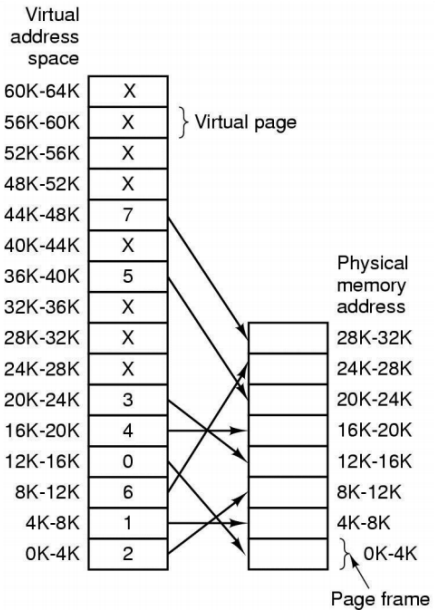

图 P4-1: 分页机制和页框

3.3 页表（page table）

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），它们保存了虚拟地址到物理地址的映射关系。图P4-2所示的就是通过页表项进行虚实地址转换的一个过程。

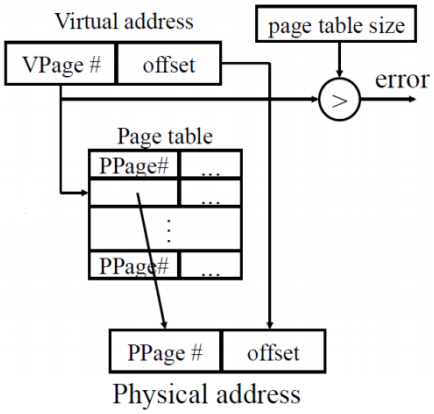

图 P4-2: 分页机制和页框

RISC-V 的 TLB 管理是硬件自动完成的，因此需要页表遵循硬件定义好的格式。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。

由于我们的物理内存只有 256MB，选择多级页表太过麻烦。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、物理地址和页表项格式如图P4-3图P4-4图P4-5所示 [1]。

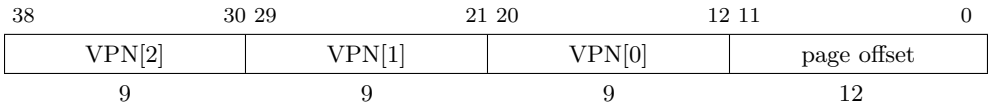

图 P4-3: Sv39 虚地址

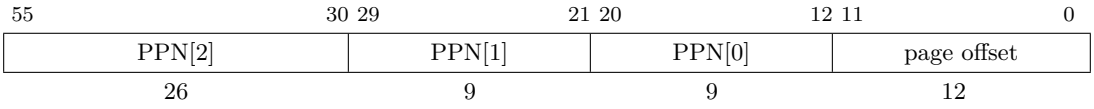

图 P4-4: Sv39 物理地址

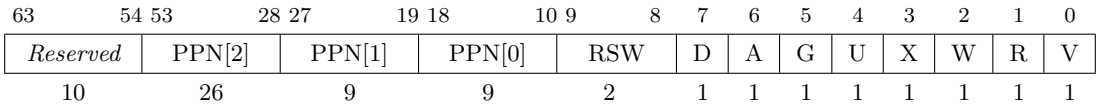

图 P4-5: Sv39 页表项

每个页可以设置一系列的属性，RISC-V 一共提供了 8 个不同的属性。其中，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。A 和 D 代表 Access 和 Dirty。用于表明页面是否被访问或者写入过。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。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。

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。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。、

3.4 SATP 寄存器及 TLB 相关指令

CSR 寄存器中的 SATP 寄存器负责放置页目录的物理地址。MMU 每次会根据这里记录的物理地址去查找对应的页目录，完成地址翻译的过程。SATP 寄存器的结构如图P4-6所示。

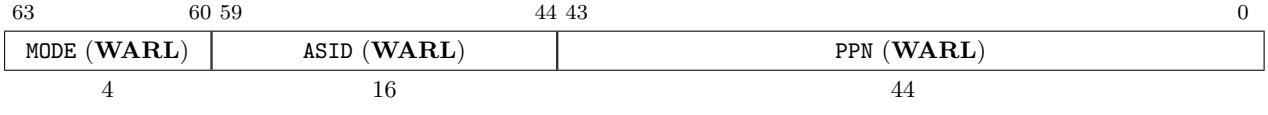

图 P4-6: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.[1]

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。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]

最后，**还需要注意一个很关键的指令，叫做 sfence.vma。这个指令是用来刷 TLB 的。因为 TLB 是硬件自动管理的，当我们希望清空 TLB 时，可以使用该指令。**另外需要注意，如果对地址映射进行了修改，必要的时候还需要用 fence 和 fence.i 刷新数据 cache 和指令 cache。

3.5 启用虚存机制进入内核

内核的页表由于是在内核加载前就需要设置好的。主要代码位于 head.S 和 boot.c 中，这部分的代码我们在 start-code 中已经给好。我们采用两级页表，相当于每个页面的大小为 2MB。

这里之所以像代码中这么实现的原因是，虚存一旦开启，则所有访问都会被认为是虚地址，因此，我们需要首先用一小段程序建立内核页表，并将内核拷贝到对应的虚地址上。之后开启虚存，再跳入到对应的位置上去。原先我们是 bootloader->kernel，在本次实验中，变为了 bootloader->head->kernel。这里为了更进一步贴近真实的系统，本次的内核将以 elf 文件的形式存放，由 head 将其加载到对应位置。

为了便于大家载入 elf 文件，我们提供了 elf.h，其中有一个 load_elf 函数。后面无论是解析内核的 ELF 还是用户的 ELF，我们都会使用这个函数。

关于 elf 文件格式的处理和 Project4 内核镜像的结构

本次实验提供了两个写好的工具，在 tools 下，分别是 elf2char 和 generateMapping。这两个工具是为了弥补我们目前没有文件系统的遗憾。elf2char 可以将一个文件的二进制读入，并输出一个 c 文件（或者头文件），文件中包含了该文件的二进制和长度。例如，makefile 中会执行，elf2char main > payload.c。在 payload.c 中就会包含类似于如下内容：

```
unsigned char _elf_main[] = {
0x7f,0x45,0x4c,0x46,0x02,0x01,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x02,0x00,0xf3,0x00,0x01,0x00,0x00,0x00,0x00,0x00,0x30,0x10,0xc0,0xff,0xff,0xff,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
// ... 此处略去
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
};
int _length_main = 248932;
```

其中，_elf_main 中的内容就是 main 文件转换成 char 数组以后的样子，该数组的内容和 main 文件完全一样。通过这种方式，我们可以把内核的 elf 文件嵌入到最后的镜像中。

目前整个内核的结构变成了：由 head.S 和 boot.c 链接在一起，形成 start 的部分，由该部分开启虚存并加载内核。start.s、main.c 等部分会编译链接在一起，形成真正的内核部分。

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 start 部分是运行在物理地址上的，开启了虚地址以后，start 的代码就也会通过虚存机制来访问内存。为了让 start 的代码不出问题（start 的代码都是在 0x50201000 这一段运行的），所以需要临时把 0x50201000 所在的 2MB 的页映射到 0x50201000 这个虚地址上，这样才能让 start 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。

另外有一个需要注意的点是，我们板子上的 sbi_sd_read 没法一次读取太多的 sector。本次的镜像编译出来可能很大，建议按照 64 个 sector 一组分多次读入。

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。这一次，我们将会把用户程序真正地编译为单独的可执行文件，并在操作系统中加载执行，拥有各自的虚地址空间。

4.1 crt0.S

在操作系统中，其实每个程序都会链接一个 crt0.o。这个 crt0 主要是用来帮助用户程序做一些必要的初始化，然后调用用户程序的 main 函数，最后退出。之前的实验中，我们每个测试程序退出时都会主动调用 sys_exit。像这样重复的动作，完全可以放在 crt0.S 中，每次写完程序，编译的时候和 crt0.S 编译到一起，到时候就会自动调用 sys_exit 啦。我们在 start-code 中也提供了这一文件。

4.2 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，exec、kill、ps、ls。其中，ls 负责显示所有可以 exec 的名字，这些名字会由 generateMapping 程序自动生成出来。

exec 命令的格式变化为 exec 可执行文件名 参数 参数...，可以有 0 个或多个参数，用空格隔开。

4.3 加载可执行文件

generateMapping 函数会解析 elf2char 生成的.h 和.c 文件，整理出所有的可执行文件的名字，然后生成一个用于根据文件名查找对应二进制的代码。其生成的代码大致如下：

```
// user_programs.h
typedef struct ElfFile {
  char *file_name;
  unsigned char* file_content;
  int* file_length;
} ElfFile;

#define ELF_FILE_NUM 3
extern ElfFile elf_files[3];
// 注意 file_name 部分的名字是由 elf2char 生成的变量名自动简化后得到的，方便在命令行里输入。
extern int get_elf_file(const char *file_name, unsigned char **binary, int *length);
// user_programs.c
#include <os/string.h>
#include <user_programs.h>
ElfFile elf_files[3] = {
  {.file_name = "shell", .file_content = _elf___test_test_shell_elf, .file_length = &_length___test_te
  {.file_name = "fly", .file_content = _elf___test_fly_elf, .file_length = &_length___test_fly_elf},
  {.file_name = "rw", .file_content = _elf___test_rw_elf, .file_length = &_length___test_rw_elf}
};
int get_elf_file(const char *file_name, unsigned char **binary, int *length)
{
  for (int i = 0; i < 3; ++i) {
    if (strcmp(elf_files[i].file_name,file_name) == 0) {
      *binary = elf_files[i].file_content;
      *length = *elf_files[i].file_length;
      return 1;
    }
  }
  return 0;
}
```

根据用户输入到 shell 中的命令，请大家加载可执行文件。

4.4 任务一：执行用户程序

实验要求

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

实验步骤

1. 为每个进程设置用户页表，要求用户进程的页的大小为 4K，采用三级页表。
2. 将内核页表映射到用户页表中（但不允许用户访问）

3. 载入 ELF 文件到相应的虚地址

4. 修改调度器，调度时切换页表

5. 修改 kill 和 exit 系统调用的实现，回收分配的物理页

注意事项

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。

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。**

另一个需要注意的点就是，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，推荐为 0xf00010000。然后映射一个物理页给这个虚地址，作为用户的内核栈。

本实验要求实现的 exec 是可以接受命令行参数的，由 shell 执行 sys_exec 时，将命令行参数传递给操作系统。系统需要将其拷贝到新打开的进程的栈上，这是一次跨地址空间的拷贝，具体怎么做到，就需要大家自己来设计了。(提示：可以利用内核虚地址和物理地址是线性映射这一关系)。

在实验中，需要注意一个细节，RISC-V 的 ABI[2] 要求栈指针的地址是 128 对齐的。前面的实验中，我们并没有严格遵循这一约定。这次实验在启用用户进程的时候建议大家遵循这一点，以免出现意料之外的问题。

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，我们只分配了用于加载可执行文件的页面和 4K 的栈空间，没有分配其他空间。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。

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/lock.c 的两个程序作为测试用例，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。但是这里同学们可以看到 lock 测试的源代码有了变化，调用锁的方式的函数名称有了变化，使用 LOCK 和 UNLOCK 作为 binsemop 的参数来操作锁。而且锁的 id 是由一个 binsemget 函数获取到的，相当于原来用 lock_init 的方式来获取锁，但是强调了锁的 id 是由内核分配的。同学们可以让对应的功能函数依然调用之前实现好的进程锁，但是要确保之前实现的版本已经保证了数据的隔离。否则在开启了虚存的情况下，可能就会触发缺页中断，即使分配了物理页面也无法和另一个进程使用同一把锁。

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 sbi 调用实现。这里需要注意的是，我们提供的 sbi 的 read 和 write 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, 如何选择哪一个物理页框进行替换需要替换算法支持, 在本任务中，请同学们设计一个替换算法 (可以参考操作系统理论课介绍的算法)，思考物理页应该如何索引，如何替换，并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。

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。

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。任务三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。这一测试本来是在 Project3 的 C-core

中由同学们完成的任务，但是有了线程之后，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/mailbox.c。

同学们需要实现的是线程相关的功能和接口，比如测试程序中的 mthread_create 等，并保证 P3 实现的 mailbox 相关系统调用正确。mthread_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 mthread_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 mthread_create 函数的接口逻辑是，该进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。原本的线程在调用 mthread_create 创建完这个线程之后执行 send_thread 作为发送线程。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内存传递这一字符串。recv_thread 和 send_thread 里面分别调用了 mbox_send，mbox_recv，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 mthread_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。由于我们测试程序的特殊性，main 函数应该不会执行到这里，recv 线程应该也不会结束。所以这个函数的功能同学们可以简化或暂时不实现。但是严谨的话还是希望同学们实现好线程回收的机制。

注意事项

要让操作系统能看到进程创建的线程并调度起来，mthread_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用户栈，以独立的运行代码和控制各自的临时变量。

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。请大家将 SUM 位置 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。

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：虚存下的 fork 和 copy-on-write 机制

C-core 的同学们在 Project2 的时候已经完成了一个 fork 的功能。当时没有完成也没有关系，因为有了虚存之后，fork 的功能会变得容易实现。即新创建的子进程会拥有自己独立的虚拟地址空间，从而很容易的与父进程独立开来。理论上子进程的创建需要把父进程的所有数据都拷贝到子进程的地址空间中，但是实际上，父进程的许多数据子进程可能都用不到。copy-on-write 机制就是来缓解这一现象所带来的额外数据拷贝开销。

copy-on-write 机制的含义是，子进程在创建之初拥有了与父进程独立的虚地址空间，但这个虚地址空间与父进程的虚地址空间完全共享相同的物理地址空间，这样就不需要进行数据的拷贝了。而这块物理地址空间被标为只读，一旦父子进程之一对这段数据内容进行了更改，就会触发 page fault，从而给子进程分配新的物理地址空间，并把原本的数据拷贝过去。拷贝完成后再由父进程或子进程对这段数据内容进行修改，从而避免数据冲突，又降低了进程建立时的数据拷贝开销。

任务五就需要同学们完成这样的 fork 功能。由同学们自行完成测试程序的编写，体现带 copy-on-write 机制的 fork 系统调用功能正确。

8.3 任务六：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/consensus.c。

多核共享内存需要实现两个接口：

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。

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P4-7所示。

图 P4-7: 共享内存测试效果

9 多进程与多线程的对比

对于 C-core 的同学来说，我们要求一直使用双核处理器进行测试。而有一个功能的性能优势只有在双核下才能明显的体现出来，那就是多线程对于多进程的性能优势。多进程的情况下两个进程如果需要通信，需要通过内核的锁或者信号量等服务来进行，申请和释放的过程开销都较大。而多线程通信可以使用多线程所属的进程资源，通过共享内存来进行通信。这样的通信机制就省去了内核的参与，开销小了许多。特别是如果两

个进程或两个线程跑在两个不同的核上时，这样的开销差距就会变的明显。因为两个核由于内核锁的存在无法同时进入内核态处理数据，而两个线程却可以通过简单的自旋锁方式完成线程的阻塞和释放。

9.1 任务七：多进程与多线程加法对比

任务七要求同学们分别用多进程机制与多线程机制完成同样的功能：对同一个数字做加法。那么在多进程情况下，同学们需要使用之前完成的进程间通信和共享内存的机制由两个进程对同一个数字进行加法运算。而在多线程情况下，同学们需要实现轻量级的线程自旋锁，来完成多线程对同一个数字的加法运算。最终同学们需要分别运行两种情况，并体现出明显的性能差距。测试程序由同学们自行设计完成。

注意自旋锁的定义时一个线程拿不到锁的时候忙等另外一个线程释放锁，即不断的轮询锁的状态，直到锁被释放时拿到锁。这时对自旋锁的操作需要使用原子指令来完成。原子指令的调用方式在 start-code 中有提供，参见 tinylibc/include/stdatomic.h。详细的 RISC-V 原子指令定义可以参见 [3]。简单而言，对于锁的操作，我们可以使用 start-code 里封装好的 atomic_exchange(volatile void* obj, int desired) 这个函数来实现，参数 desired 表示一个要交换给参数 obj 的值。比如如果一个线程要加锁，那就要把一个 1 交换给 lock 变量，然后通过返回值判断交换出来的值是 0 还 1，如果交换出来的是 0，则加锁成功；如果交换出来的是 1，则说明锁本来就是锁上的，则加锁失败，该线程没有获得到锁。

参考文献

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

[2] e. Sam Elliott, “Risc-v elf psabi specification.” https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md, 2019. [Online; accessed 25-August-2021].

[3] “The risc-v instruction set manual volume i: User-level isa v2.2,” 2017.