# 国科大操作系统研讨课任务书

# Project 4


版本：2.1

# 目录


一、 实验说明 ..... 3

二、 任务一：页框、页表和 TLB ..... 5

2.1. 页框（page frame） ..... 5
2.2. 页表（page table） ..... 6
2.3. 初始化 TLB ..... 7
2.3.1. 龙芯处理器中的 TLB 项 ..... 7
2.3.2. TLB 操作中使用的相关寄存器 ..... 7
2.3.3. TLBWI 指令 ..... 9
2.4. 任务一要求 ..... 9
2.5. 测试用例 ..... 9

三、 任务二：TLB 的例外处理 ..... 11

3.1. TLB 重填例外和 TLB 无效例外 ..... 11
3.2. TLB 修改例外 ..... 12
3.3. 用户栈的设置 ..... 12
3.4. 任务二要求 ..... 12
3.5. 测试用例 ..... 12

四、 任务三：缺页、按需调页和内存隔离 ..... 13

4.1. 缺页（page fault） ..... 13
4.2. 按需调页（On-demand Paging） ..... 13
4.3. 内存保护 ..... 13
4.4. 任务三要求 ..... 13
4.5. 测试用例 ..... 13

五、 Bonus：swap 和页替换算法 ..... 15

5.1. 换页机制（page swap） ..... 15

# 一、 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，TLB 的应用，换页机制等。与之前的实验相比，本实验的代码量较多，但是除了 TLB 例外处理的任务，其他任务都是纯软件管理的部分，实现和调试难度较小，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：

**任务一：初始化页框、页表和 TLB**

1、 将物理地址空间划分为页框

2、 设计页表项，为页表分配内存空间

3、 初始化一个静态的页表，并在初始化时设置好 TLB 项

**任务二： TLB 的例外处理**

1、 学习龙芯处理器上 TLB 的相关指令，完成 TLB 例外处理

2、 实现进程的用户态栈使用 mapped 内存空间

**任务三：缺页、按需调页和内存隔离**

1、 实现缺页处理程序，初始化时并不建立虚实地址映射，只在发生缺页中断时才分配物理页

2、 实现内存隔离机制，使得一个进程访问其他进程的内存数据时系统报错

**Bonus：换页机制**

1、 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

2、 实现页替换算法

一次虚存访问的流程如图 1 所示，其中各个任务需要完成的部分已经用不同颜色的框表示出来。

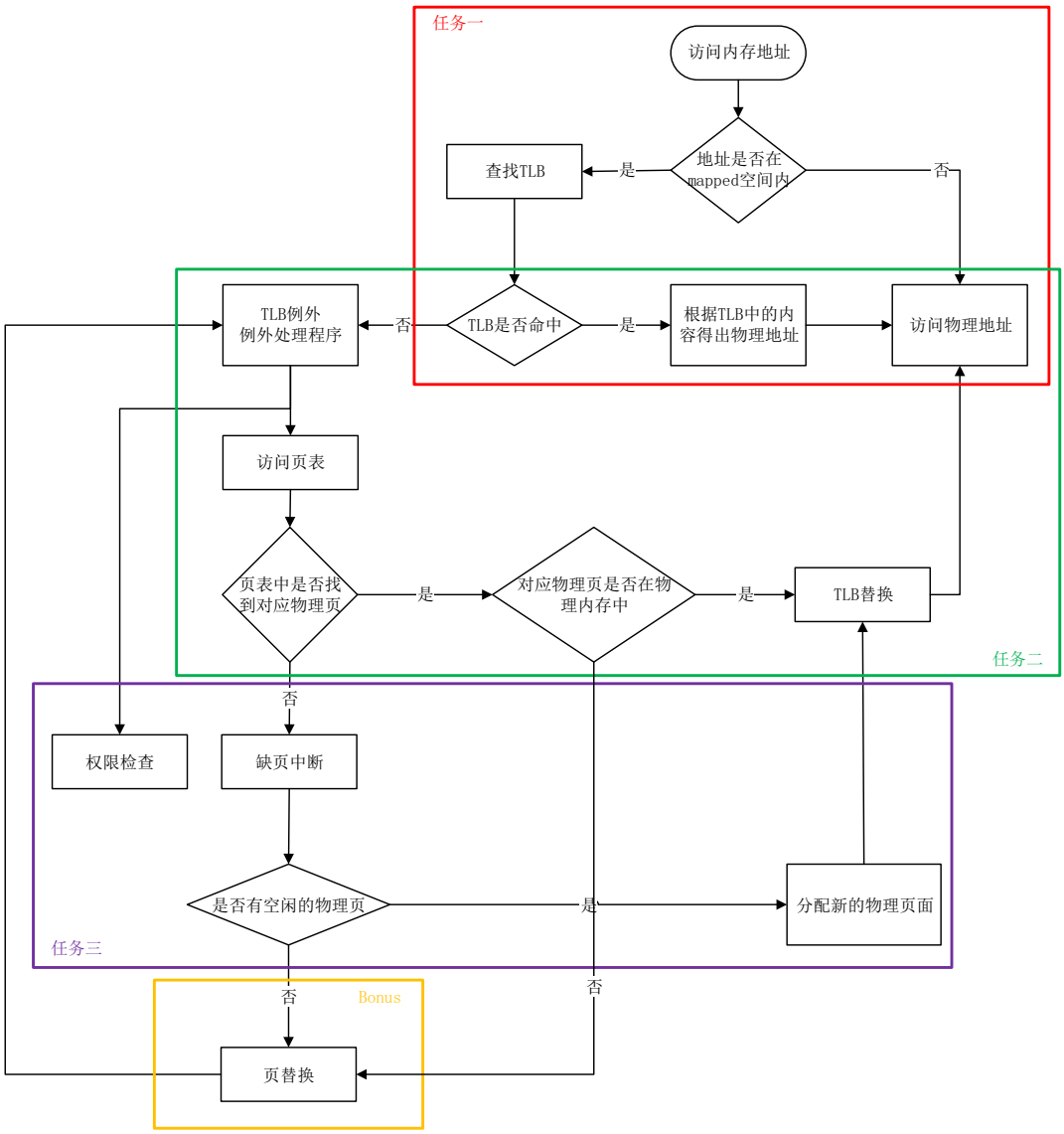



图 1 Project 4 流程图

请同学们在阅读完本任务书之后再参考此图进行虚存管理部分的设计。

和 Project2、3 一样，我们提供了一个初始的代码框架 start_code，里面会给出一些要实现的基本函数，同学们可以参考它，在它的基础上完成本实验。注意：请将本次 project 的 start-code 增加到同学们自己实现好的 project3 中，进而继续增加虚存管理功能，Makefile 文件可以直接替换使用本次 project 的 Makefile，也可以阅读理解 Makefile 后，根据自己增加的 c 文件自行增加 SRC_MM 部分。当然，同学们也可以通过合理的设计，改变现有的代码框架，也许你能设计出更好的虚存管理机制。

除了本任务书之外，同学们可以参考《龙芯 2F 处理器手册》中的内容，以使得操作系统代码的内容可以符合硬件的约定，任务书中摘录了手册中大部分相关的内容，但是依然推荐同学们在学习和调试的过程多参考手册。特别是，由于龙芯 2F 处理是 64 位处理器，但是我们开发板上的龙芯 1C 处理器是 32 位的，所以手册上与任务书中不一致的地方以任务书为准。同时，同学们也需要结合理论课学到的知识，设计出一套完备的虚存管理机制。从下一章开始，我们将按任务的顺序，详细介绍同学们应该完成的功能。

# 二、 任务一：页框、页表和 TLB

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 0xa0800000 的位置，而在调用读取 SD 卡的函数时，需要将函数指针设为 0x80011000。不仅如此，同学们在调试中可能也会偶尔发生系统报告 TLB exception 的错误，报这样的错是因为访问到了 0x80000000~0xC0000000 之外的地址。我们可以通过图 2 来回顾一下开发板上的内存地址空间分布，标识为 Mapped 的地址区间访问是不会直接转换为物理地址，而是会通过查 TLB 的方式进行转换，而同学们之前没有对 TLB 进行过处理，就会陷入在 TLB 例外中无法正常运行。

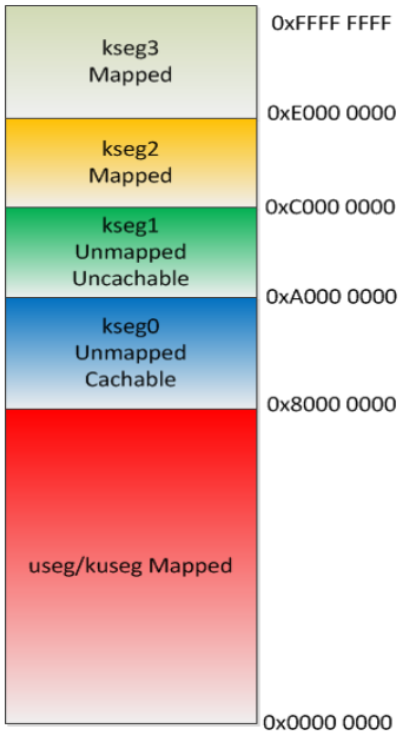



图 2 开发板上的内存空间

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。

## 2.1. 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 3 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。

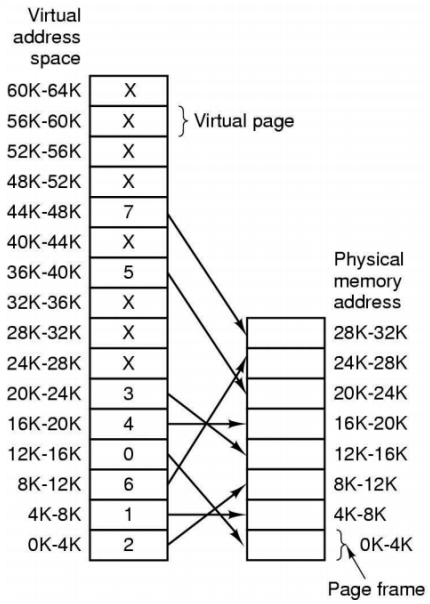



图 3 分页机制和页框

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？

## 2.2. 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 4 所示的就是通过页表项进行虚实地址转换的一个过程。

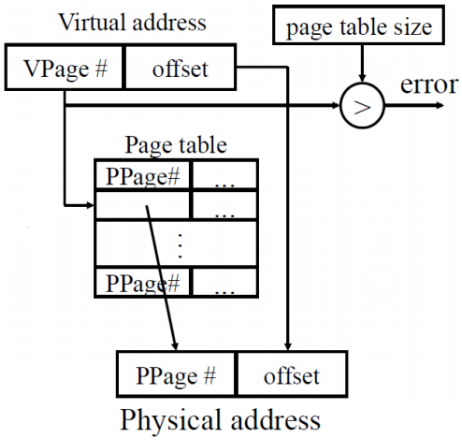



图 4 页表的工作方式

每一个页表项中存储的内容应能表达一个从虚拟页号到物理页号的映射关系，所以它**至少应该包含虚拟页号、物理页号（page frame number）和有效位（valid flag）的信息**。请同学们思考，页表项中还需要包含哪些信息，页表需要放在内存中的哪部分空间，需要占多大的空间？页表的索引方式如何实现？在本实验中，页表的索引方式请同学们自由实现，并考虑索引效率，有兴趣的同学可以实现多级页表。

## 2.3. 初始化 TLB

从这一小节的内容开始，推荐各位同学们参考《龙芯 2F 处理器手册》中的内容，主要是第 3 章内存管理、第 8 章 8.2 节 TLB 控制指令、第 5 章中和内存管理相关的 CP0 寄存器以及第 6 章中 TLB 例外的相关章节。任务书会贴出相关要用到的内容，但是为了同学们更确定龙芯处理器对相关环节的约定，请同学们在编写自己的代码前最好通读相关的手册内容。

在之前的实验中，我们通过直接访问物理地址的方式使用了从 0x80000000~0xC0000000 的地址空间，而如果我们的代码出错，访问到这段地址空间之外的地址，就会触发 TLB 例外。而在之前的实验中，我们没有实现 TLB 例外，因此访问这些地址就会失败。而本实验中，为了把其他的地址空间用起来，我们就需要实现 TLB 例外。而在任务一中，我们先在初始化时预先填充 TLB，从而在使用中不触发 TLB 例外。

### 2.3.1. 龙芯处理器中的 TLB 项

在我们使用的龙芯处理器中，TLB 是软件管理的，即我们需要使用 TLB 操作指令来完成 TLB 项的填充。而在任务一中，我们只需要 TLB 的填充指令，即 **TLBWI 指令**。TLBWI 指令的格式是 TLBWI（即后面没有指令参数）。对 TLBWI 指令的描述是：*用 EntryLo0 和 EntryLo1 寄存器的 G 位相与的值设置 TLB 的 G 位。用 EntryHi 和 EntryLo 寄存器的内容设置 Index 的值所索引的 TLB 表项。如果 TLB Index 寄存器的值大于处理器中的 TLB 表项数，则该操作无效。* 可能这样看它的描述完全无法理解，但是在此请同学们注意：**使用 TLBWI 指令对 TLB 项进行填充之前，必须先设置好 EntryHi、EntryLo、Index 这几个寄存器**。于是，我们先看一下这几个寄存器和 TLB 项的格式。

在介绍这几个寄存器之前，我们先看一下龙芯处理器中 TLB 项包含的内容，如图 5 所示。

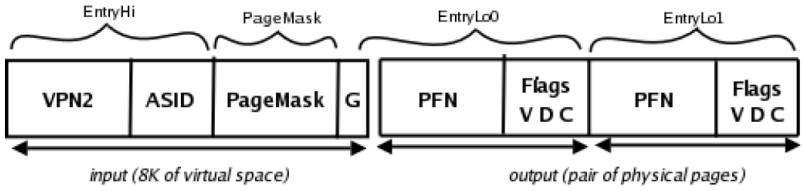


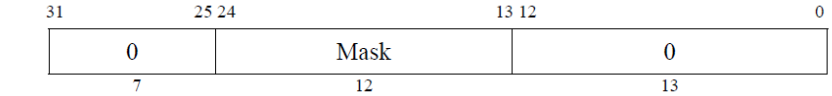

图 5 TLB 项格式

从图 5 中可以看到，龙芯处理器 TLB 项包含两个 PFN 域，这是因为在这个 TLB 项的设计中，**一个 TLB 项将相邻的两个虚拟页（奇数页和偶数页）映射到了两个对应的物理页**。而 TLB 项其他域的含义，我们将在接下来介绍相关的寄存器进一步描述，因为它们都在相关的寄存器中出现了。

### 2.3.2. TLB 操作中使用的相关寄存器

首先我们要介绍的是 PageMask、EntryHi 和 EntryLo 这三个寄存器。

（1） PageMask 寄存器：




*Mask*……页比较掩码

*0*……保留。写入必须是 0，读时返回 0。

图 6 PageMask 寄存器

图 6 所示的是 PageMask 寄存器的格式，可以看到它的内容和图 5 中的 PageMask 相对应。在该寄存器中，除了 Mask 域之外，其他的内容都是 0。在图 6 中，Mask 域的含义是指页比较掩码，那么什么是页比较掩码呢？我们先直接来看图 7。

| 页大小 | 位 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 |
| 4Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 64 Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 256 Kbytes | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 Mbytes | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 4 Mbytes | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 16M bytes | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

图 7 不同页大小的 Mask 值

图 7 规定的是 Mask 可能的取值，图 7 所允许的内容之外的其他取值都视为无效。我们举例说明 Mask 域的含义：如果操作系统中的页面大小是 4KB，那么一个 32 位的虚拟地址中 1~12 位都是页内偏移位，其余的高位地址都作为页号。即在匹配两个地址的页号时，我们只需要比较 13~32 位的地址即可；但是如果页面大小为 16KB，一个 32 位的虚拟地址中 1~14 位都是页内偏移位，我们在匹配两个地址的页号时，只需要比较 15~32 位的地址即可。因此，对应图 7 我们可以理解到，**Mask 域的含义就是指示了当前的页面大小，同时指示了每次进行页号比较的时候，在一个 32 位的虚拟地址中，需要比较的起始地址位**。这就是 Mask 域的含义。

PageMask 寄存器的内容用于设置 TLB 项中的 PageMask 域。因此，TLB 项中的 PageMask 域也是相同的含义。**如果同学们采取统一 4KB 大小的页面，Mask 域的值永远是 0。**如果页面大小不是 4KB，请同学们对照图 7 设置相应的 PageMask 寄存器的值。

（2） EntryHi 寄存器：

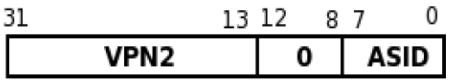



图 8 EntryHi 寄存器

图 8 所示的是 EntryHi 寄存器的内容，它对应了图 5 TLB 项中的 VPN2 和 ASID 域。VPN2 域表示一对连续的虚拟页号，**请注意，在 32 位的虚拟地址空间中，如果使用 4KB 大小的页面，虚拟页号应该是 13~31 位，在此处填充 VPN2 域的值时，需要进一步把虚拟页号除 2，即去掉虚拟地址的第 13 位，从而表示一对连续的页号**。ASID 域表示进程的进程 ID 号，**在任务一中，这个域的内容暂时不做要求，ASID 统一设置为 0 即可。请注意：在后续任务中，当需要区分与不同进程相对应的 TLB 项时，ASID 域须设置为对应进程的进程号。**

EntryHi 寄存器的 VPN2、ASID 域的含义与 TLB 项中完全相同。

（3） EntryLo0 和 EntryLo1 寄存器：

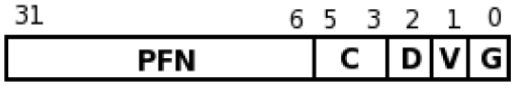



图 9 EntryLo 寄存器

图 9 所示的是 EntryLo 寄存器的内容，它对应了图 5 TLB 项中的 EntryLo0 和 EntryLo1 域,，其中 EntryLo0 用于偶数虚页，而 EntryLo1 用于奇数虚页。PFN 域表示的是物理页号。C 域指定 TLB 页的一致性属性，D 域表示脏位（实际表示是否可写），V 域是有效位，G 域是全局位，具体的含义请同学们查阅手册。**注意：在任务一中，永远设置为 C 的值是 2，D 的值是 1，V 的值是 1，G 的值是 1 即可。**

至此，我们通过介绍相关的寄存器，也介绍了 TLB 项中所有域的内容，请同学们掌握上述 TLB 项中的内容及其对应的寄存器。

剩下还有一个寄存器会在 TLB 操作中使用，即 Index 寄存器。

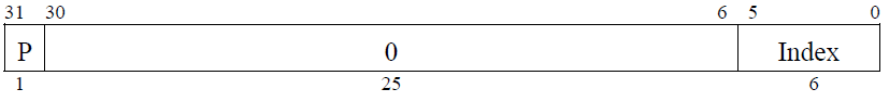



图 10 Index 寄存器

图 10 表示了 Index 寄存器的格式，其中 Index 域指示 TLB 读指令和 TLB 写指令操作的 TLB 表项的索引值；P 域是探测失败域，**注意：当执行 TLB 探测指令（TLBP）没有成功时该位置 1，在任务一中，P 域暂时不会用到。**

### 2.3.3. TLBWI 指令

通过 2.3.1 和 2.3.2 节的介绍，我们已经了解了 TLB 操作中会使用到的 TLB 表项和相关寄存器的格式。这时我们可以介绍 TLBWI 指令的使用了。我们改写了《龙芯 2F 处理器手册》中提供的一个代码例子来介绍：

```
li k1, (vpn2<<13)|(asid & 0xff)
mtc0 k1, C0_ENHI                    # 设置虚拟页面
li k1, (epfn<<6)|(coherency<<3)|(Dirty<<2)|Valid<<1|Global)
mtc0 k1, C0_ENLO0                   # 设置偶数物理页面
li k1, (opfn<<6)|(coherency<<3)|(Dirty<<2)|Valid<<1|Global)
mtc0 k1, C0_ENLO1                   # 设置奇数物理页面
li k1, 0                            # 设置页面大小为 4KB
mtc0 k1,C0_PAGEMASK
li k1, index_of_some_entry          # 设置 TLB 表项的索引值
mtc0 k1, C0_INDEX
tlbwi
```

请同学们认真学习上述代码示例，可以看到，一次通过 TLBWI 指令进行 TLB 写的逻辑就如上述代码例子所示，**先将相关的寄存器都设置好，最后调用 TLBWI 指令即可**，而相关寄存器的含义我们在 2.3.2 节中已经介绍过了，同学们可以再用代码例子对照着看一遍。

至此，基本的 TLB 项、内存管理寄存器和 TLBWI 指令的用法已经介绍完毕。

## 2.4. 任务一要求

在初始化的时候，在页表和 TLB 中填好映射项，使得测试中不会触发 TLB 例外，同时也不会触发缺页。但是我们要求这里向 TLB 中填入的项是从页表中读出来而非错误的项。后续从任务二开始，我们再逐步实现一个真正的 TLB、TLB 例外以及缺页处理程序。

注意：原先的 PMON 和操作系统内核代码所在的物理地址为虚拟地址去掉高位的地址。例如 0xa0800000 对应物理地址 0x800000。请同学们在选择要映射的物理地址时避免与 PMON 代码或操作系统内核代码相冲突。

## 2.5. 测试用例

process1.c 和 process2.c 是两个测试用进程，可以使用 shell 上的 exec 命令启动。process1.c 的内容只是普通的打印飞机。而 process2.c 的进程需要六个输入，前三个输入分别输入一个虚拟地址，process2 会往输入的三个虚拟地址里写入随机数（随机数在 process2 中由系统时间生成）；后三个输入也分别输入一个地址，process2 会读出输入的虚拟地址中的数据。例如测试时，同学们可以在前三次输入虚拟地址 0x100000、0x120000 和 0x200000，后三次也输入 0x100000、0x120000 和 0x200000，如果写入的随机数和读出的数据相同，测试程序会输出”succeed!”，即为虚存读写成功。所以在任务一时，我们需要在初始化时将 process2.c 中所需要的虚实地址映射建立好，具体虚址和物理地址范围请自行定义，并同时填充 TLB 的内容。**注意：由于 TLB 项数共为 32 项，所以在任务一中，静态填充 TLB 的项数不能超过 32 项，因此，任务一中所对应的页面数也不能覆盖所有地址空间，即我们只能映射一部分虚实地址。**

请同学们修改 Project3 里面 exec 命令的实现，使它可以带上 6 个参数，以满足测试用例 process2.c 里面 rw_task1

的输入参数需求。

# 三、 任务二：TLB 的例外处理

在任务一中，我们已经建立了一个虚实地址映射的机制。但是在 TLB 部分，我们使用的是静态填充，而非动态填充和替换的方式。在任务二中，我们就要来实现一个完整的 TLB 功能，其核心就是 TLB 例外的处理。

TLB 例外一共有三种情况：TLB 重填例外（refill）、TLB 无效例外（invalid）和 TLB 修改例外，同学们需要分别处理前两种情况。在 project2 中，我们已经实现了时钟中断的处理函数，并且知道中断的例外码，即 Cause 寄存器的 ExcCode 域的值是 0，而对于 TLB 例外，我们需要加入图 11 中这三个例外的例外处理流程。

| 例外代码 | Mnemonic | 描述 |
|---|---|---|
| 0 | INT | 中断 |
| 1 | MOD | TLB 修改例外 |
| 2 | TLBL | TLB 例外（读或者取指令） |
| 3 | TLBS | TLB 例外（存储） |

图 11 Cause 寄存器的 ExcCode 域

对于时钟中断，例外的向量入口地址是 0x80000180。而对于 TLB 例外，向量入口地址是 0x80000000，也就是说，对于 TLB 例外，硬件会自动跳转到 0x80000000 这个地址，然后再交给软件进行处理，执行我们编写的例外处理代码。

任务书会给出每种例外的介绍和处理流程，同学们也可以参考《龙芯 2F 处理器手册》第 6.8 节至第 6.11 节的介绍。

## 3.1. TLB 重填例外和 TLB 无效例外

这一节将介绍 TLB 重填例外和 TLB 无效例外，之所以将这两种例外放在一起，是因为他们的处理流程是几乎相同的。

TLB 重填例外是指 TLB 中没有项与欲引用的地址映射匹配。发生 TLB 重填例外时，ExcCode 的值是 TLBL 或 TLBS（即 2 或 3），TLBL 表示引发这个例外的指令是取操作指令，而 TLBS 表示引发这个例外的指令是存操作指令。

发生 TLB 重填例外之后，BadVAddr、Context、EntryHi 寄存器都存储了触发例外的对应虚拟地址，而我们实现的例外处理代码中，我们应该根据 Context 寄存器中的内容（即虚拟地址）来访问页表，得到对应的物理地址之后，将虚实地址对填充进 TLB 中。填充 TLB 的方法已经在任务一中介绍过了。

当虚地址引用与 TLB 中某一项匹配，但该项被标记为无效时，TLB 无效例外发生。TLB 无效例外的 ExcCode 依然是 TLBL 或 TLBS（即 2 或 3）。而处理流程也几乎和 TLB 重填例外相同，都是需要根据 Context 寄存器中存储的虚拟地址，从内存中找出对应的物理地址。**注意：唯一与 TLB 重填例外不同的是，填充 TLB 项时不应该随意填充一个位置，而是应该找出与虚拟地址匹配但是标记为无效的那一项，并用有效的一项来替换它。**所以在填充 TLB 项之前，我们应该使用 TLBP 指令来找出匹配但是无效的那一项的 index 值。

TLBP 指令的使用示例如下：

*mtc0 a1, CP0_EntryHi*

*tlbp*

将 a1 寄存器设置正确（设置虚地址和 ASID），并写入 EntryHi 寄存器，TLBP 指令就会自动根据 EntryHi 寄存器中的内容来找到 TLB 中对应的项，并将项的 index 值存入 Index 寄存器中，如果没有找到，则 Index 寄存器的 P 域将置为 1。

至此，同学们应该可以看出，两个例外的处理代码是一份代码，只是最后调用 TLBP 时，TLB 无效例外会找到一个项，而 TLB 重填例外找不到项。

### 3.2. TLB 修改例外

这一节将介绍 TLB 修改例外，当写内存指令操作的虚地址引用与 TLB 中某项匹配，但该项并没有被标示为“脏”，则表明该虚址对应的内存页面不可写。此时，如果对该页面执行写操作，则 TLB 修改例外发生。TLB 修改例外的 ExcCode 是 MOD（数值是 1）。

在龙芯处理器中，TLB 项的 D 域就是脏位标识，它表示的并不是相应页面有没有被改写过，而是指相应页面是否能改写，充当写保护位的作用。在本实验中，我们不会出现页面不能改写的情况，也就是说在各种情况下，**D 域都应该是 1 而不是 0**，因此 TLB 修改例外应该不会触发。但在这里还是说明这个例外，主要是告诉同学们，如果你在调试过程中发生了 TLB 修改例外，那么一定是你某次填充 TLB 项时，D 域没有置位正确。或者你也可以实现一个 TLB 修改例外，每次将对应 TLB 项的 D 域置位为 1，以确保可以改写页面。

### 3.3. 用户栈的设置

在应用了 TLB 例外处理程序之后，我们已经可以完全使用 mapped 的地址空间（参加图 2），这样，我们可以把程序的用户栈分配在用户空间。即初始化 PCB 时，用户栈的地址将从 0x00000000~0x7FFFFFFF 中分配。

请同学们思考如何完成这样的功能。

### 3.4. 任务二要求

1）仅初始化页表，不初始化 TLB 项，实现 TLB 重填例外和无效例外的例外处理；2）将测试进程的用户栈分配在 mapped 的地址空间（即 0x00000000~0x7FFFFFFF），使进程能正常运行。

### 3.5. 测试用例

任务二依然使用任务一中的 process1.c 和 process2.c 作为测试用例，测试方法与任务一相同，区别在于：在任务二中，process2 的虚拟地址可以任意指定。在任务二中，初始化时，我们只需要初始化页表，而不需要初始化 TLB 内容了，在实际访问数据时通过 TLB 重填例外来填充 TLB 项。而且，由于整个页表都可以按需加载进 TLB 中，因此所有地址空间都可以使用，即 process2.c 中用于测试的输入地址可以在完整地址空间范围（例如，0x00000000~0x7FFFFFFF）中选择（注意：需要在页表中已将相应映射关系初始化），但是不能超出物理地址空间。

# 四、 任务三：缺页、按需调页和内存隔离

在任务一和任务二中，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。

在任务三中，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。

## 4.1. 缺页（page fault）

缺页处理程序可以是 TLB 例外处理程序的一部分，因为在我们的实验中，如果一个被访问的虚拟页面找不到它对应的物理页面，它的 TLB 项一定也在 TLB 中无法找到。因此，在我们的 TLB 例外处理程序需要将正确的 TLB 项填充进 TLB 时，如果它在页表中找不到对应的页表项时，就进入缺页处理程序。

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，我们将在任务三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 bonus，希望同学们也能实现。

## 4.2. 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，初始化的时候不再建立虚实映射的页表项，只是分配好页表所需的空间。而在 TLB 例外处理程序中，同学们需要实现：如果一个虚拟页面到物理页面的映射无法在页表中找到，就建立一个映射，并分配物理页框，并将映射关系写入页表项，最后填充进 TLB。

## 4.3. 内存保护

在前面的任务完成之后，我们实现了分页机制、TLB 例外、缺页处理，大家已经可以正确地使用虚拟地址来访问内存，但是这里存在一个问题，如果一个进程访问了不属于它的地址空间，我们还没有机制来阻止它。所以在任务三中，我们还要实现这种内存隔离机制。

对于内存的保护与隔离，我们主要实现进程之间的隔离：即一个进程访问其他进程虚址数据时报错退出，打印报错信息。注意：TLB 项中有 ASID 域，可以用来表示不同进程的地址空间，如果硬件访问 TLB 时虚地址匹配但是 ASID 不匹配（例如进程 A 访问虚址 0x1000，TLB 中有 0x1000 的映射，但是并不是进程 A 的映射），系统也会发生 TLB 重填例外。基于该例外，可以实现进程间的地址空间隔离，请同学们设计并实现该种内存保护方式。

## 4.4. 任务三要求

初始化页表时仅设置虚拟地址，不建立虚实映射，并且不填充 TLB，自行实现缺页处理程序，动态建立页表映射，分配物理页框，填充 TLB 项，同时实现进程间的地址空间隔离。

## 4.5. 测试用例

到了任务三，我们依然使用 process1.c 和 process2.c 的内容，测试方法与任务二相同。注意：加入了按需调页

机制后，初始化时将不会分配物理页面。

# 五、 Bonus：swap 和页替换算法

在完成了前三个任务之后，我们已经可以使用全部的内存空间和 TLB 项，并实现了 TLB 例外处理程序和按需调页机制。Bonus 的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。

## 5.1. 换页机制（page swap）

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。

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。所以，我们首先介绍为同学们提供的 SD 卡读写库。

我们给同学们提供了一个代码中可以调用的静态库，即 libepmon.a，将它们和我们的操作系统代码放在一起，并在 Makefile 中的编译命令中加上-L. -lepmon 就可以调用它们了。例如：

*mipsel-linux-gcc -G 0 -O0 -fno-pic -mno-abicalls -fno-builtin -nostdinc -mips3 -Ttext=0xffffffffa0800200 -N -o kernel kernel.c -L. -lepmon -nostdlib -e main -Wl,-m -Wl,elf32ltsmip -T ld.script*

而在这些静态库中，我们提供了 SD 卡的读写函数 sdread 和 sdwrite，它们的函数定义分别如下：

*void* sdread*(unsigned char *buf, unsigned int base, int n)*

*void* sdwrite*(unsigned char *buf, unsigned int base, int n)*

其中 buf 是内存中的一块地址，base 是 SD 卡上开始读写的位置，n 是读写的长度（单位是 512Byte 的扇区），请同学们注意，base 请选择扇区对齐的地址，内核在初始化时需要定义好 swap 的起止地址，并在使用过程中对其进行管理。

通过这两个函数，我们就可以实现 SD 卡的读写了。请同学们熟悉一下这两个函数用法，后续的 project 中还将有用到它们的时候。

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。

注意：系统触发缺页中断时，如果需要进行换页，则会进行磁盘 I/O，这将是一个比较耗时的操作。因此，一个程序在进入换页操作时，应该将自己挂起（block），等待换页动作完成。因此，换页机制**推荐**设计成如下的流程：

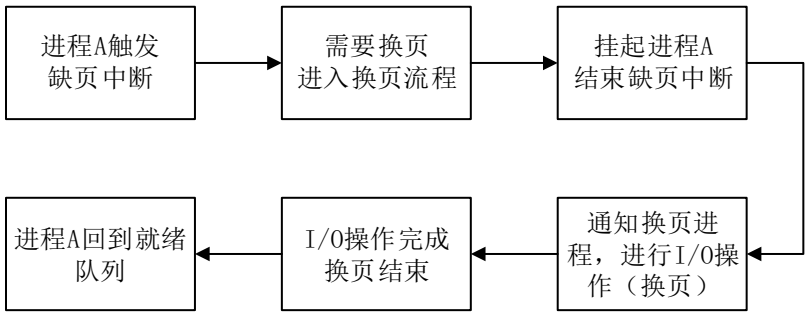



图 12 换页流程

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 bonus 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循环遍历操作，从而可以触发页替换。如果要测试替换算法有效性，虚址遍历操作时需要有热点页面的访问。