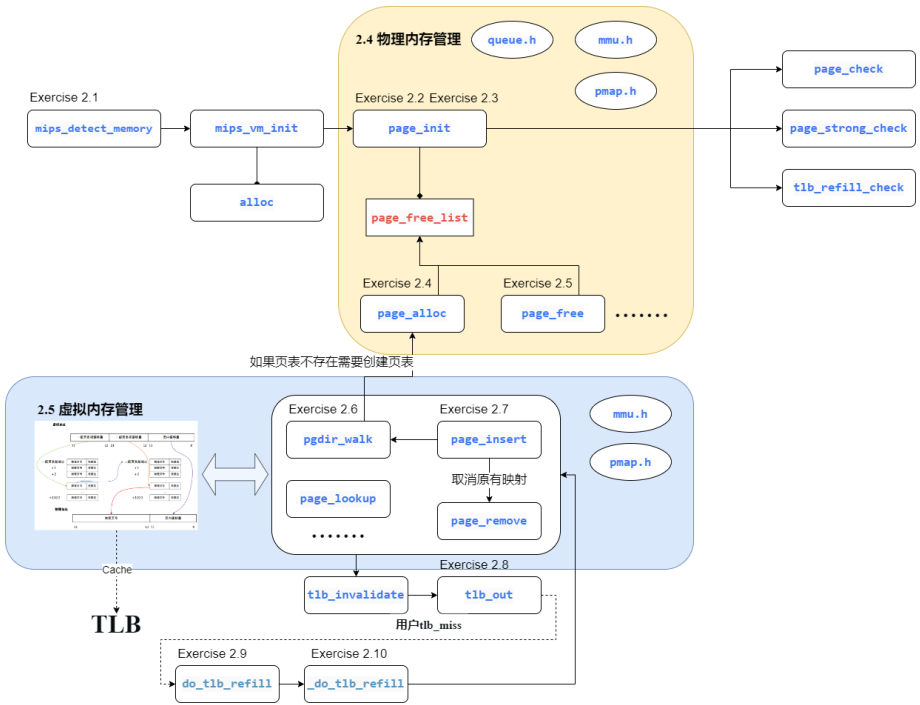
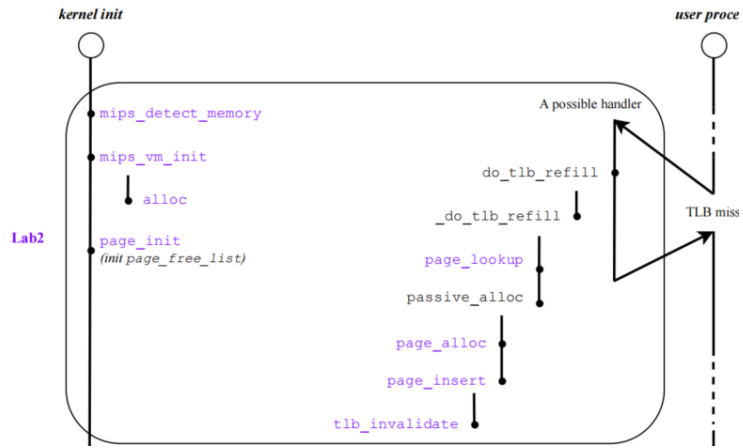
2023操作系统 Lab2 串讲

2023操作系统助教组

Lab2 整体调用关系


2.4 物理内存管理
queue.h
mmu.h
pmap.h
Exercise 2.1
mips_detect_memory
mips_vm_init
alloc
Exercise 2.2 Exercise 2.3
page_init
page_check
page_strong_check
tlb_refill_check
page_free_list
Exercise 2.4
page_alloc
Exercise 2.5
page_free
如果页表不存在需要创建页表
2.5 虚拟内存管理
Exercise 2.6
pgdir_walk
page_lookup
Exercise 2.7
page_insert
取消原有映射
page_remove
mmu.h
pmap.h
Cache
TLB
Exercise 2.8
tlb_invalidate
tlb_out
用户tlb_miss
Exercise 2.9
do_tlb_refill
Exercise 2.10
_do_tlb_refill


kernel init
user proce
Lab2
mips_detect_memory
mips_vm_init
alloc
page_init
(init page_free_list)
A possible handler
do_tlb_refill
_do_tlb_refill
page_lookup
passive_alloc
page_alloc
page_insert
tlb_invalidate
TLB miss

Lab2 中地址相关的常用宏

在 `pmap.h`、`mmu.h` 中：

- `PDX(va)`：页目录偏移量（查找遍历页表时常用）
- `PTX(va)`：页表偏移量（查找遍历页表时常用）
- `PTE_ADDR(pte)`：获取页表项中的物理地址（读取 pte 时常用）
- `PADDR(kva)`：kseg0 处虚地址 → 物理地址
- `KADDR(pa)`：物理地址 → kseg0 处虚地址（读取 pte 后可进行转换）
- `va2pa(Pde *pgdir, u_long va)`：查页表，虚地址 → 物理地址（测试时常用）
- `pa2page(u_long pa)`：物理地址 → 页控制块（读取 pte 后可进行转换）
- `page2pa(struct Page *pp)`：页控制块 → 物理地址（填充 pte 时常用）

内核程序启动与初始化

exercise2.1 `mips_detect_memory`：探测硬件可用内存

- 我们的初始化的最初阶段：探测硬件可用内存。内存大小由 GXemul 启动命令决定，可从 GXemul 模拟的 mp 外设中读取内存大小。
- 同时对一些和内存管理相关的变量进行初始化。包括：

1. `memsize`，表示总物理内存对应的字节数。
2. `npage`，表示总物理页数，此处可以思考 memsize 与 npage 的关系，并参考 `PGSHIFT` 用法。

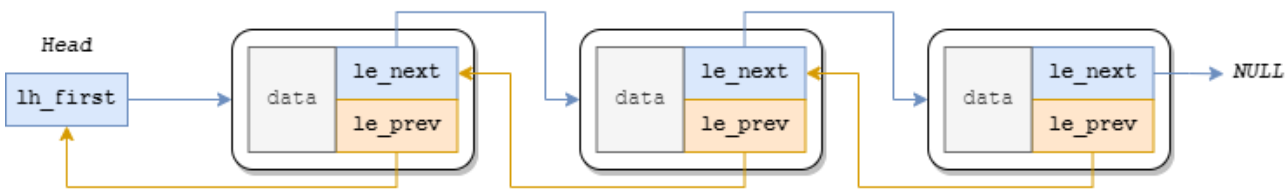
`mips_vm_init`：给页控制块数组分配空间

- 在探测完可用内存后，`mips_vm_init()` 将开始建立内存管理机制。为了建立起内存管理机制，还需要用到 `alloc` 函数，在没有页式内存管理机制时，操作系统也需要建立一些数据结构（即页控制块 `Page` 数组 `pages`）来管理内存，这就会涉及到内存空间的分配。`alloc` 函数的功能就是用于分配内存空间（在建立页式内存管理机制之前使用）。

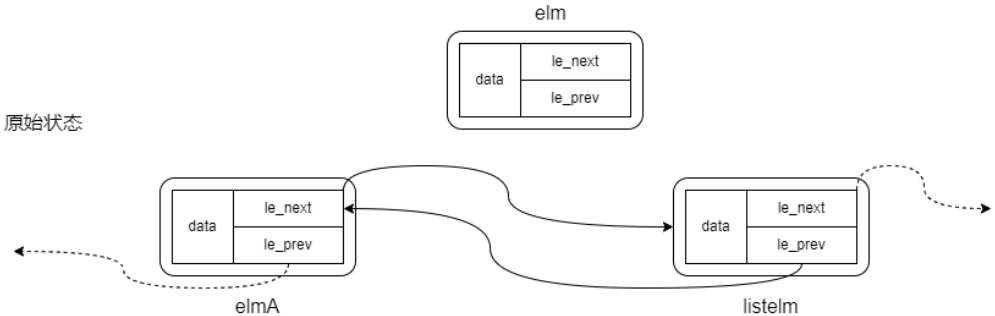
物理内存管理

exercise2.2 exercise2.3 物理内存管理之`page_init`：用链表串起空闲页控制块

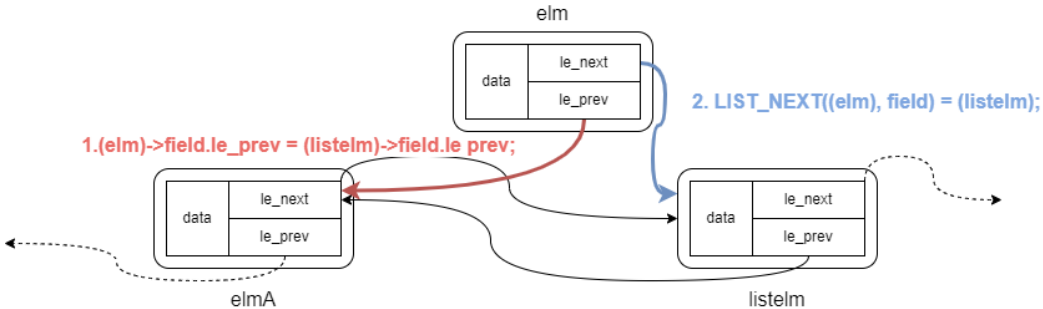
- 我们用链表 `page_free_list` 来管理空闲物理页面。而这需要我们编写 `queue.h` 中的链表宏。在 `include/pmap.h` 中可以看到若干个以 `page_` 开头的函数。其中，有关物理内存管理的函数有 4 个，分别用来初始化物理页面管理 `page_init`、分配物理页面 `page_alloc`、减少物理页面引用 `page_decref`、回收物理页面到空闲页面链表 `page_free`。



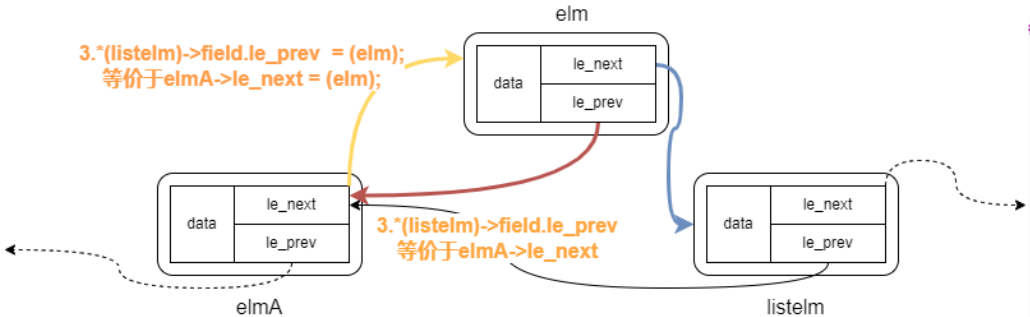
原始状态



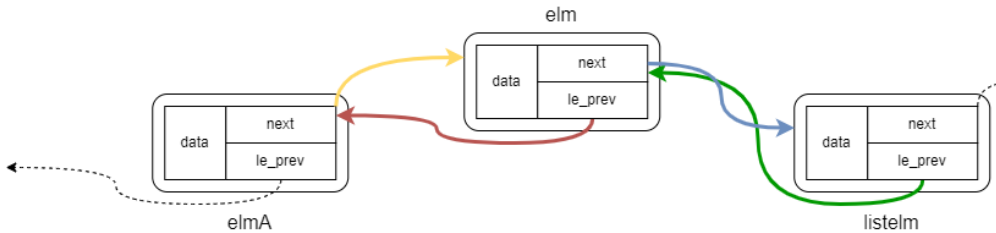

```
#define LIST_INSERT_BEFORE(listelm, elm, field)                 \
    do {                                                        \
        (elm)->field.le_prev = (listelm)->field.le_prev;        \
        LIST_NEXT((elm), field) = (listelm);                    \
        *(listelm)->field.le_prev = (elm);                      \
        (listelm)->field.le_prev = &LIST_NEXT((elm), field);    \
    } while (0)
```




```c
#define LIST_INSERT_BEFORE(listelm, elm, field)                         \
    do {                                                                \
        (elm)->field.le_prev = (listelm)->field.le_prev;                \
        LIST_NEXT((elm), field) = (listelm);                            \
        *(listelm)->field.le_prev = (elm);                              \
        (listelm)->field.le_prev = &LIST_NEXT((elm), field);            \
    } while (0)
```



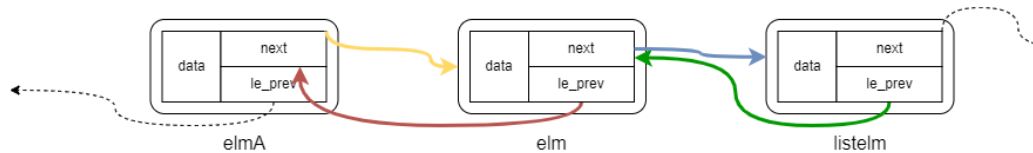

```c
#define LIST_INSERT_BEFORE(listelm, elm, field)                 \
    do {                                                        \
        (elm)->field.le_prev = (listelm)->field.le_prev;        \
        LIST_NEXT((elm), field) = (listelm);                    \
        *(listelm)->field.le_prev = (elm);                      \
        (listelm)->field.le_prev = &LIST_NEXT((elm), field);    \
    } while (0)
```



4. (listelm)->field.le_prev = &LIST_NEXT((elm),field);

```
#define LIST_INSERT_BEFORE(listelm, elm, field)                 \
    do {                                                        \
        (elm)->field.le_prev = (listelm)->field.le_prev;        \
        LIST_NEXT((elm), field) = (listelm);                    \
        *(listelm)->field.le_prev = (elm);                      \
        (listelm)->field.le_prev = &LIST_NEXT((elm), field);    \
    } while (0)
```

整理一下




```
#define LIST_INSERT_BEFORE(listelm, elm, field)                 \
    do {                                                        \
        (elm)->field.le_prev = (listelm)->field.le_prev;        \
        LIST_NEXT((elm), field) = (listelm);                    \
        *(listelm)->field.le_prev = (elm);                      \
        (listelm)->field.le_prev = &LIST_NEXT((elm), field);    \
    } while (0)
```

物理内存管理

exercise2.4 物理内存管理之`page_alloc`：分配物理页面

- `page_alloc(struct Page **pp)` 它的作用是将 `page_free_list` 空闲链表头部页控制块对应的物理页面分配出去，将其从空闲链表中移除，并清空对应的物理页面，最后将 `pp` 指向的空间赋值为这个页控制块的地址。

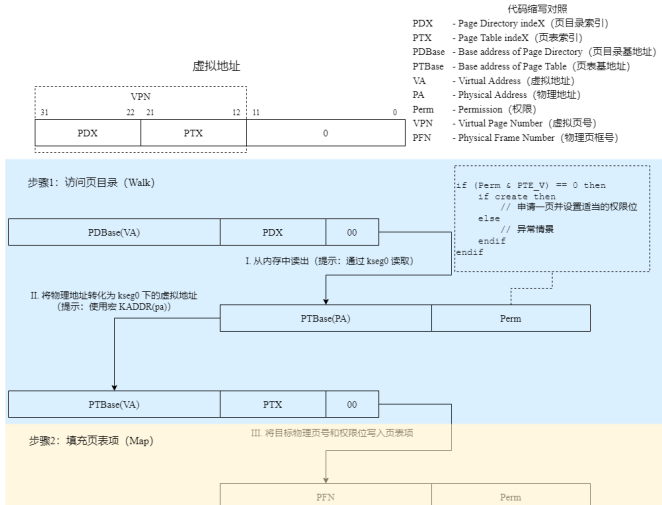
exercise2.5 物理内存管理之`page_free`：释放物理页面

- `page_free(struct Page *pp)` 调用该函数的前提条件为 `pp` 指向页控制块对应的物理页面引用次数为 0（即该物理页面为空闲页面）。其行为为将其对应的页控制块重新插入到 `page_free_list`。

虚拟内存管理

exercise2.6 `pgdir_walk`：二级页表检索函数

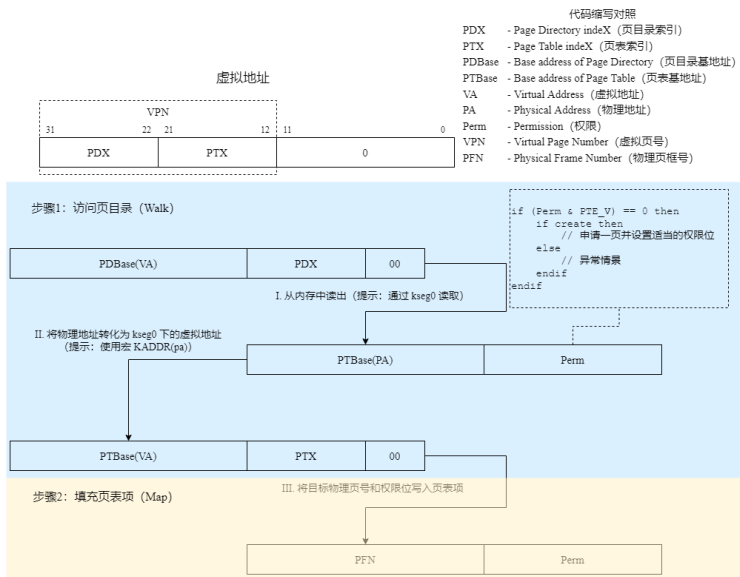
- 该函数的作用是：给定一个虚拟地址，在给定的页目录（一级页表基地址）中查找这个虚拟地址对应的物理地址，如果这一虚拟地址对应的页目录项存在（也即页目录项 `PTE_V` 位为 1），则继续访问页目录项对应二级页表，返回虚拟地址对应页表项的地址；如果虚拟地址对应的页目录项无效、不存在（也即页目录项 `PTE_V` 位为 0,那么就不存在这一页目录项对应的二级页表），则根据传入的参数或创建二级页表，或返回空指针。



虚拟内存管理

exercise2.7 `page_insert`：增加地址映射函数

- `int page_insert(Pde *pgdir, u_int asid, struct Page *pp, u_long va, u_int perm)`，作用是在页目录 `pgdir` 对应的两级页表结构中将虚拟地址 `va` 映射到页控制块 `pp` 对应的物理页面，并将页表项权限设置为 `perm`。



TLB 的维护与访存流程

- Lab2 还未涉及到用户进程相关的内容，所有代码、数据的虚拟地址均在 kseg0 段，无需通过页表的翻译便可直接获得其物理地址。因此本次实验中所完成的代码，大多是为之后的实验提供接口。

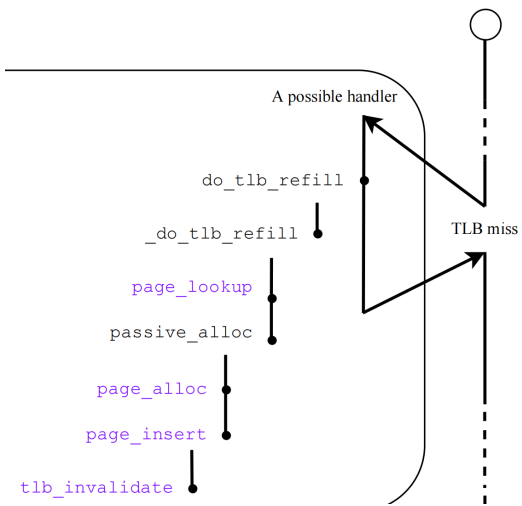
- 在用户地址空间（kuseg）访存时，虚拟地址到物理地址的转换均通过 TLB 进行。访问需要经过地址转换的内存时，首先要使用虚拟页号和当前进程 ASID 在 TLB 中查询其对应的物理页号（如果 EntryLo 的 G 位为高，则 CPU 发出的虚拟地址只需要与该表项的 VPN 匹配，不需要检查 ASID 是否匹配），如果能够查询到，则可取得物理地址；如果不能查询到，则产生 TLB Miss 异常，跳转到异常处理程序中（Lab2 只实现了 TLB 重填的处理程序，并不能由硬件直接跳转至该处理程序，Lab3 将实现完整的异常处理流程），在页表中找到对应的物理地址，由我们编写的软件对 TLB 进行重填。本实验所使用的 MIPS-R3000 的硬件只提供了 TLB，需要由软件实现上述的 MMU 机制。

- 为了保证 TLB 作为页表的缓存，其内容与页表始终一致，我们需要在更新页表中已存在的页表项的同时，调用 `tlb_invalidate` 函数删除 TLB 中对应的旧表项，使得在下一次访问该虚拟地址时触发 TLB 重填异常，由内核对 TLB 进行重填，避免旧的 TLB 项被使用。

exercise2.8 TLB 旧表项无效化：`tlb_out`

- 我们通过 `tlb_invalidate` 调用 `tlb_out` 实现删除特定虚拟地址在 TLB 中的旧表项。使得在更新页表后，用户在访问相应地址时，能够即时发生 TLB Miss，后进行页表查找，对 TLB 进行重填，保证 TLB 的内容与页表一致。

exercise2.9,2.10 TLB 重填：`do_tlb_refill`，`_do_tlb_refill`



- 从 BadVAddr 中取出引发 TLB Miss 的虚拟地址。从 EntryHi 的 6 – 11 位取出当前进程的 ASID。
- 以虚拟地址和 ASID 为参数，调用 `_do_tlb_refill` 函数。该函数是 TLB 重填过程的核心，其功能是根据虚拟地址和 ASID 查找页表，返回包含物理地址的页表项。（注意函数调用，保存现场）
- 将物理地址存入 EntryLo，并执行 `tlbwr` 将此时的 EntryHi 与 EntryLo 写入到 TLB 中。

debug的一些思路

解决 warning

解决基本的编译器 warning

printk 打点

或者高级一点，定义相关宏

```
// include/debugk.h
#ifndef _DBGK_H_
#define _DBGK_H_
#include <printk.h>
#define DEBUGK // 可注释

#ifdef DEBUGK
#define DEBUGK(fmt, ...) do { printk("debugk::" fmt, ##__VA_ARGS__); } while (0)
#else
#define DEBUGK(...)
#endif
#endif // !_DBGK_H_
```

debug的一些思路

printk 打点

使用宏

```
#include <debugk.h>
...
DEBUGK("checkpoint%d\n", 1);
```

GXemul 查看 CP0 的 epc

1. `r, 0` 查看 CP0 中 EPC 值
2. `objdump` 中查找 EPC
3. 排查问题