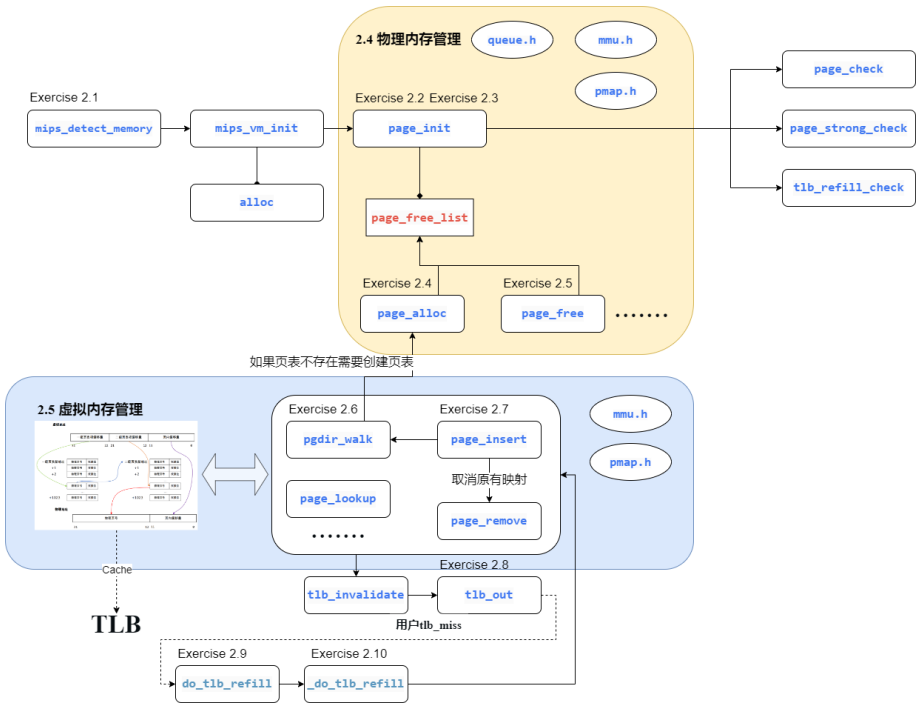
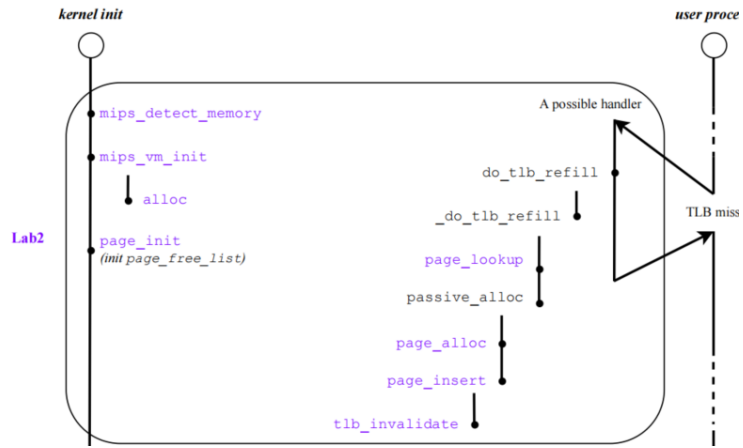
2024 操作系统 Lab2 串讲

2024 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 `include/pmap.h`、`include/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

探测硬件可用内存

在 `init/start.S` 的最后，我们跳转到了 MOS 的初始化函数 `mips_init`。本次实验我们从 `mips_init` 开始继续内核的初始化过程。

初始化的最初阶段：探测硬件可用**物理**内存。物理内存大小可由 QEMU 启动命令决定。在 `mips_init` 中的内存大小参数 `_memsize` 由 bootloader 传入操作系统。

```
void mips_detect_memory(u_int _memsize);
```

内存初始化阶段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。

内核程序启动与初始化

为页控制块数组分配空间

探测可用内存后，`mips_vm_init()` 将开始建立内存管理机制。这需要建立一些用于管理内存的数据结构。但是有一个问题，我们想要建立管理内存的数据结构，就需要内存管理已经存在。

因此在建立正式的内存管理机制之前，我们还要建立起一套临时的内存管理机制。这就是 `alloc` 函数。

```
void *alloc(u_int n, u_int align, int clear);
```

`alloc` 函数的功能就是用于在建立页式内存管理机制之前分配内存空间。使用该函数，我们为管理空闲物理页面的数据结构：页控制块数组 `struct Page *pages` 分配所用的内存空间。

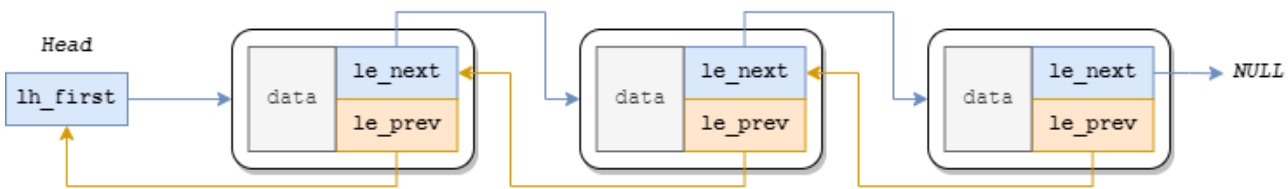
物理内存管理

内核链表

页控制块数组中的每一项都代表了一页物理内存。为了实现内存的管理，我们需要能够**高效地**实现空闲页控制块的**申请和释放**。

支持这一功能的数据结构就是**内核链表**。该数据结构由一系列宏实现，见于 `include/queue.h` 中。这一头文件实际复制自 Linux 的源码。

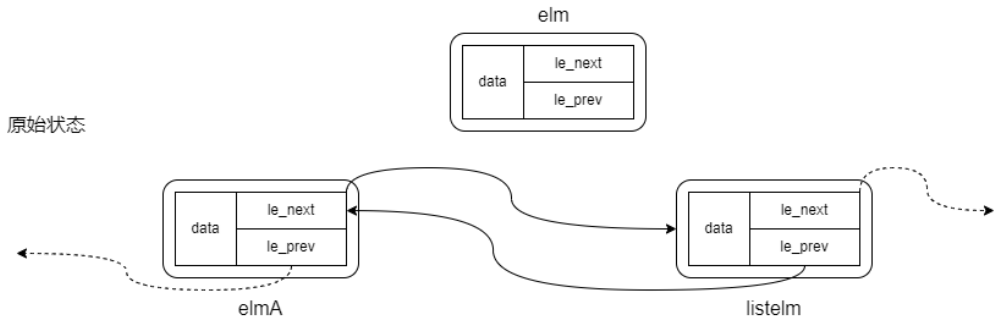
内核链表是一个双向链表，但和一般的链表有着些许不同。在内核链表中，是链表的前后指针作为数据结构体的字段，而非链表节点结构体中包含了数据字段。



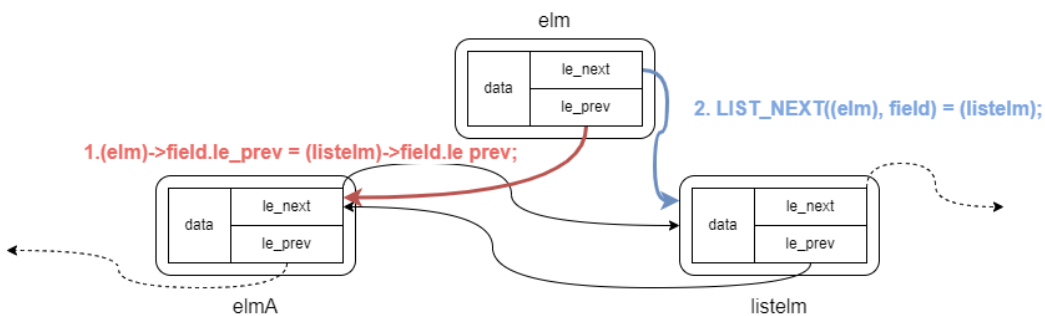
我们用链表 `page_free_list` 来管理空闲物理页面。这需要我们补全 `include/queue.h` 中的链表宏。

物理内存管理

内核链表



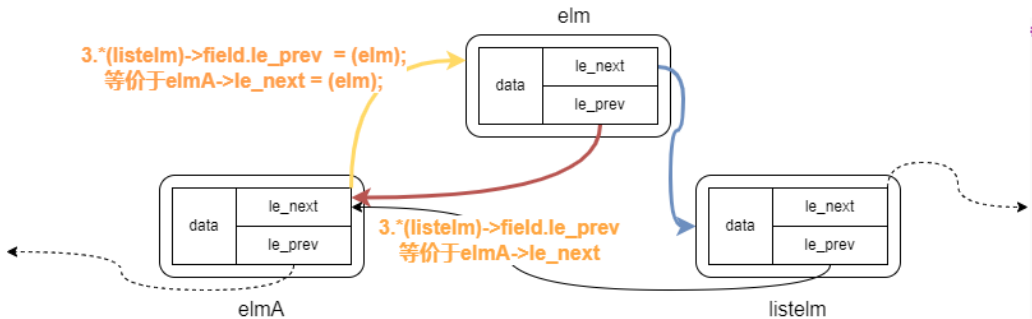

```
#define LIST_INSERT_BEFORE(listelm, elm, field)                 \
    do {                                                        \
        (elm)->field.le_prev = (listelm)->field.le_prev;        \
        LIST_NEXT((elm), field) = (listelm);                    \
        *(listelm)->field.le_prev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



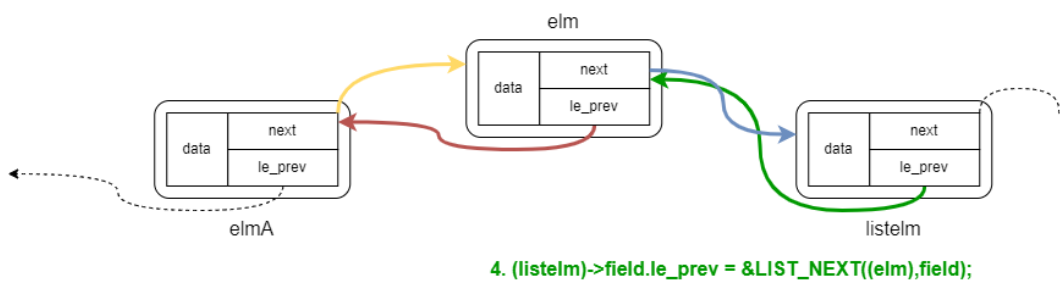

```
#define LIST_INSERT_BEFORE(listelm, elm, field)                 \
    do {                                                        \
        (elm)->field.le_prev = (listelm)->field.le_prev;        \
        LIST_NEXT((elm), field) = (listelm);                    \
        *(listelm)->field.le_prev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

物理内存管理

内核链表




```
#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)
```

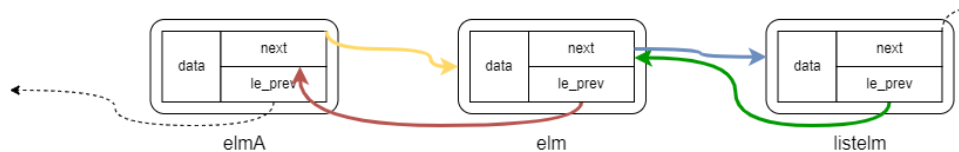



```
#define LIST_INSERT_BEFORE(listelm, elm, field)                 \
    do {                                                        \
        (elm)->field.le_prev = (listelm)->field.le_prev;        \
        LIST_NEXT((elm), field) = (listelm);                    \
        *(listelm)->field.le_prev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

物理内存管理

内核链表

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

物理内存管理函数

使用内核链表，我们建立了管理物理页面分配的数据结构 `page_free_list` 。在此基础上我们定义物理内存管理函数。这时，链表 `page_free_list` 可视为一个**资源池**。

- `page_init(void)`

  用于初始化 `page_free_list` 。包括初始化链表，初始化各页控制块，并将尚未分配的物理页对应的页控制块插入链表中。

- `page_alloc(struct Page **pp)`

  用于分配物理页面。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。

物理内存管理

物理内存管理函数

- `page_decref(struct Page *pp)`

  用于减少物理页面引用数，当引用数降为 0 时释放该物理页。

- `page_free(struct Page *pp)`

  用于将其对应的页控制块重新插入到 `page_free_list`。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）。

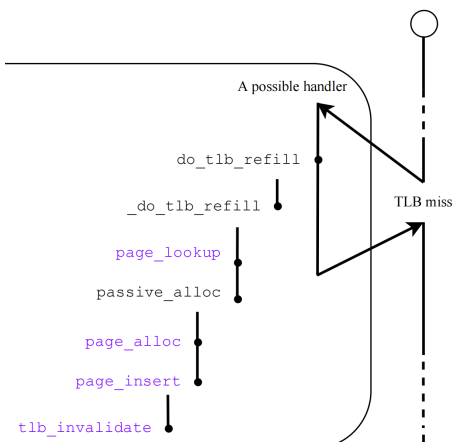
虚拟内存管理

kuseg 下的页式内存管理

当使用 kuseg 地址空间的虚拟地址访问内存时，CPU 会通过 TLB 将其转换为物理地址。当 TLB 中查询不到对应的物理地址时，就会触发 TLB Miss 异常。这时将跳转到异常处理函数，执行 TLB 重填。

在 Lab2，我们的代码还未启用异常处理，也还未涉及到用户进程相关的内容，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。

页表重填的程序入口为 `do_tlb_refill`，位于 `kern/tlb_asm.S` 中。

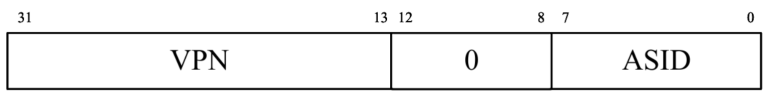


虚拟内存管理

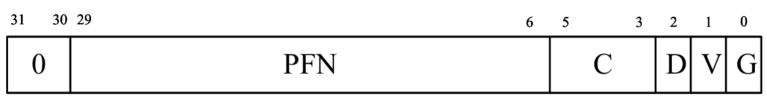
kuseg 下的页式内存管理

`do_tlb_refill` 的主要流程如下：

- 从 BadVAddr 中取出触发 TLB Miss 的虚拟地址。从 EntryHi 的后 8 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（注意在调用该函数前，需要保存现场）。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（奇偶页，两项）。
- 将物理地址存入 EntryLo，并执行 `tlbwr` 将此时的 EntryHi 与 EntryLo 写入到 TLB 中。



EntryHi Register (TLB Key Fields)



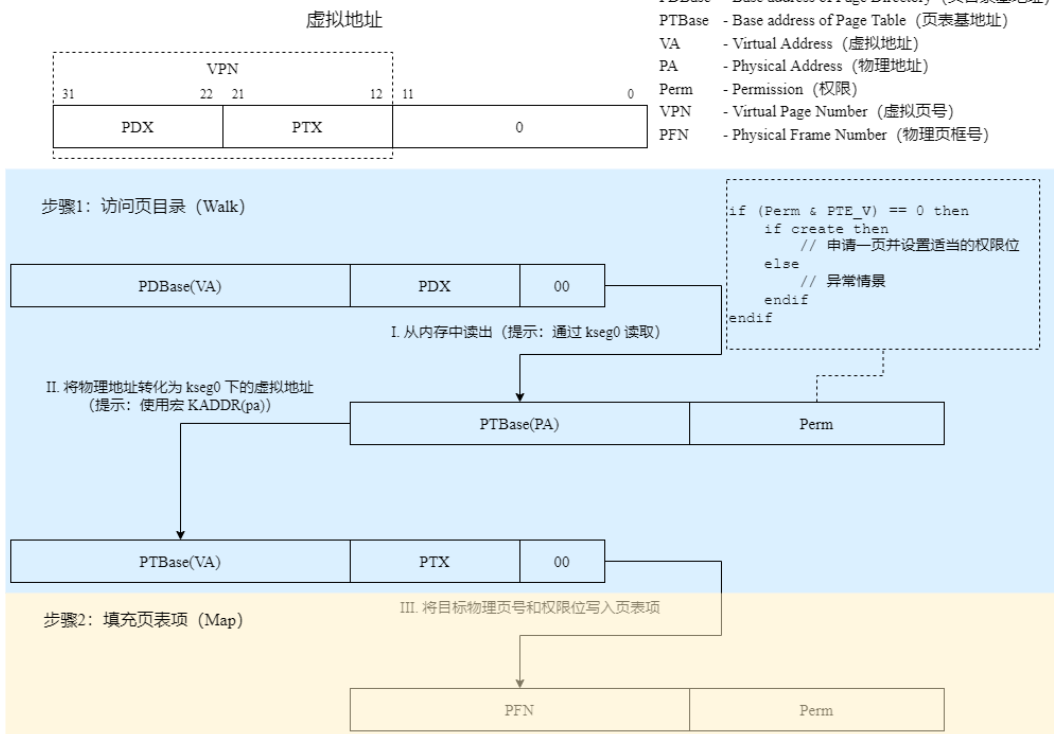
EntryLo Register (TLB Data Fields)

虚拟内存管理

walk & insert

代码缩写对照

PDX	- Page Directory indeX（页目录索引）
PTX	- Page Table indeX（页表索引）
PDBase	- Base address of Page Directory（页目录基地址）
PTBase	- Base address of Page Table（页表基地址）
VA	- Virtual Address（虚拟地址）
PA	- Physical Address（物理地址）
Perm	- Permission（权限）
VPN	- Virtual Page Number（虚拟页号）
PFN	- Physical Frame Number（物理页框号）



虚拟内存管理

walk & insert

- `pgdir_walk`：页目录检索

  `_do_tlb_refill` 中调用了 `page_lookup` 函数查找导致 TLE Miss 的虚拟地址所映射的页表项。而在 `page_lookup` 中又调用了 `pgdir_walk`（page directory walk）。

  `pgdir_walk` 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。

- `page_insert`：增加地址映射

  `_do_tlb_refill` 中调用了 `passive_alloc` 函数为虚拟地址申请物理页。其中调用了 `page_insert` 函数用于将虚拟地址和物理页相关联。

  `page_insert` 函数的作用是在页目录 `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`。

虚拟内存管理

walk & insert

- `tlb_out`：旧表项无效化

  为了保证 TLB 的内容与页表始终一致，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异常，由内核进行重填。

  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。

Debug 的一些思路

- **解决 Warning**

  解决基本的编译器 Warning

- **GDB 调试**

  - 方法一：`make dbg`（QEMU 与 GDB 的输出交织在一起，难以与 MOS 进行交互）

    1. 使用 `make test lab=<x>_<y> && make dbg` 进入某一测试点的调试环境。
    2. 使用 `tb mips_init` 和 `c` 指令进入初始化函数 `mips_init`。
    3. 使用调试指令进入到测试函数中。
    4. 经过更新，目前 GDB 退出后会自动清理 QEMU 进程，因此**只需用 `quit` 指令退出 GDB 即可**。

  在 GDB 中可以使用 `info registers` 指令查看寄存器信息。

Debug 的一些思路

- **GDB 调试**
  - 方法二：`make dbg_pts`（可与 MOS 交互，在 Lab6 很有用）
    1. 首先使用 `tmux` 分屏。
    2. 一个会话执行 `make test lab=<x>_<y> && make dbg_pts` 进入 GDB 界面。
    3. 另一个会话执行 `make connect` 通过 `screen` 连接输出。
    4. 之后操作与 `make dbg` 时相同。
    5. `screen` 退出时需要按住 `ctrl`，再分别按下 `a` 和 `d` 键。

（更详细的调试方法请同学们参考预习教程《GDB：程序的解剖术》）

Debug 的一些思路

- **Printk**

  并不推荐。但是在一些情况下还是有用的。

  或者高级一点，也可定义相关宏：

```
// 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**

  使用宏

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

课下习题提示

- Exercise 2.1

  注意理解 `PGSHIFT` 的含义

- Exercise 2.2

  1. 参考 PPT 中对 `LIST_INSERT_BEFORE` 的讲解
  2. 注意区分 `le_next` 和 `le_prev` 所指向的对象

- Exercise 2.3

  注意 `alloc` 所分配的空间不能加入到 `page_free_list` 中

- Exercise 2.4-2.5

  使用 `LIST_REMOVE` 、 `LIST_INSERT_HEAD` 实现页控制块的分配和释放

课下习题提示

- Exercise 2.6

  1. 通过 `PDX` 宏获取页目录项索引，使用 `PTX` 宏获取二级页表索引
  2. 使用 `PTE_V` 位判断二级页表是否存在（有效），需要用到位运算
  3. 注意维护页控制块的引用计数 `pp_ref`
  4. 使用 `PTE_ADDR` 宏可以获取页表项中记录的物理地址，再使用 `KADDR` 宏可将其转换为虚拟地址

- Exercise 2.7

  1. 可使用 `try` 宏向上抛出异常
  2. 使用 `page2pa` 可获取页控制块的物理地址

- Exercise 2.8-2.10

  根据指导书提示编码即可