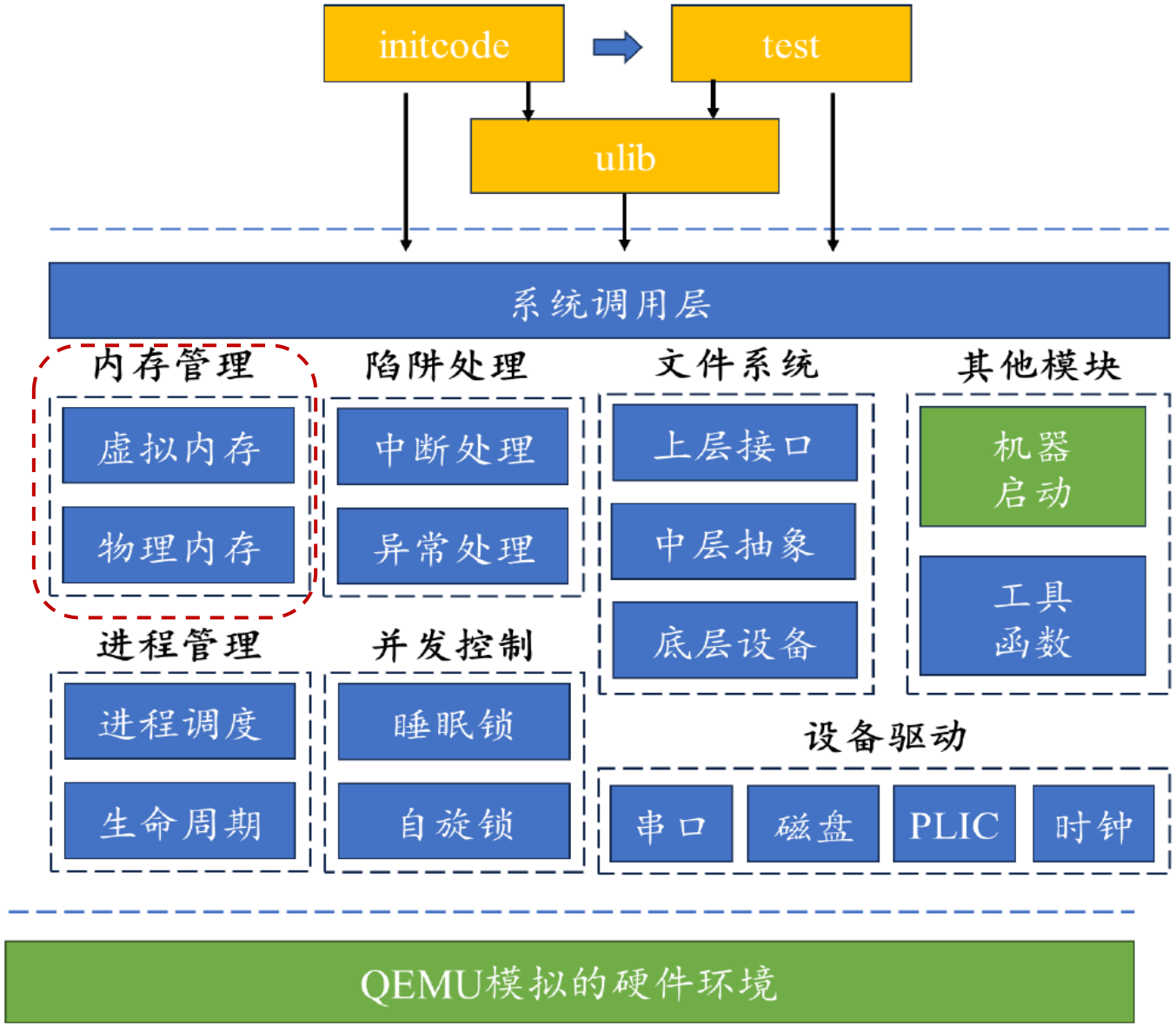
实验二：页表与内存管理



- RISC-V体系结构规定了三种特权模式：用户模式（User mode，U-mode）、监管者模式（Supervisor mode，S-mode）、机器模式（Machine mode，M-mode）。

实验二：页表与内存管理

- 本阶段目标：在上一阶段的基础上，实现对物理内存的管理，对内核构建页表，并让所有CPU启用页表。本阶段不产生额外的输出，但启用页表后，只要程序执行不错乱，仍可把提示语输出到屏幕上，就基本说明本阶段任务正确完成。

实验二：页表与内存管理

改变部分：

1. main.c 这个文件每次lab基本都会变化
2. riscv.h 最底下的与虚拟内存相关的部分删去
3. memlayout.h 新增一些硬件寄存器地址
4. common.h 新增PGSIZE定义
5. kernel.ld 新增一些PROVIDE用于标记关键地址

增加部分：增加一个子目录mem

str.h pmem.h kvm.h

str.c（lib） pmen.c kvm.c

实验二：页表与内存管理

改变部分：

1. main.c 这个文件每次lab基本都会变化
2. riscv.h 最底下的与虚拟内存相关的部分删去
3. memlayout.h 新增一些硬件寄存器地址
4. common.h 新增PGSIZE定义
5. kernel.ld 新增一些PROVIDE用于标记关键地址

增加部分：增加一个子目录mem

str.h pmem.h kvm.h

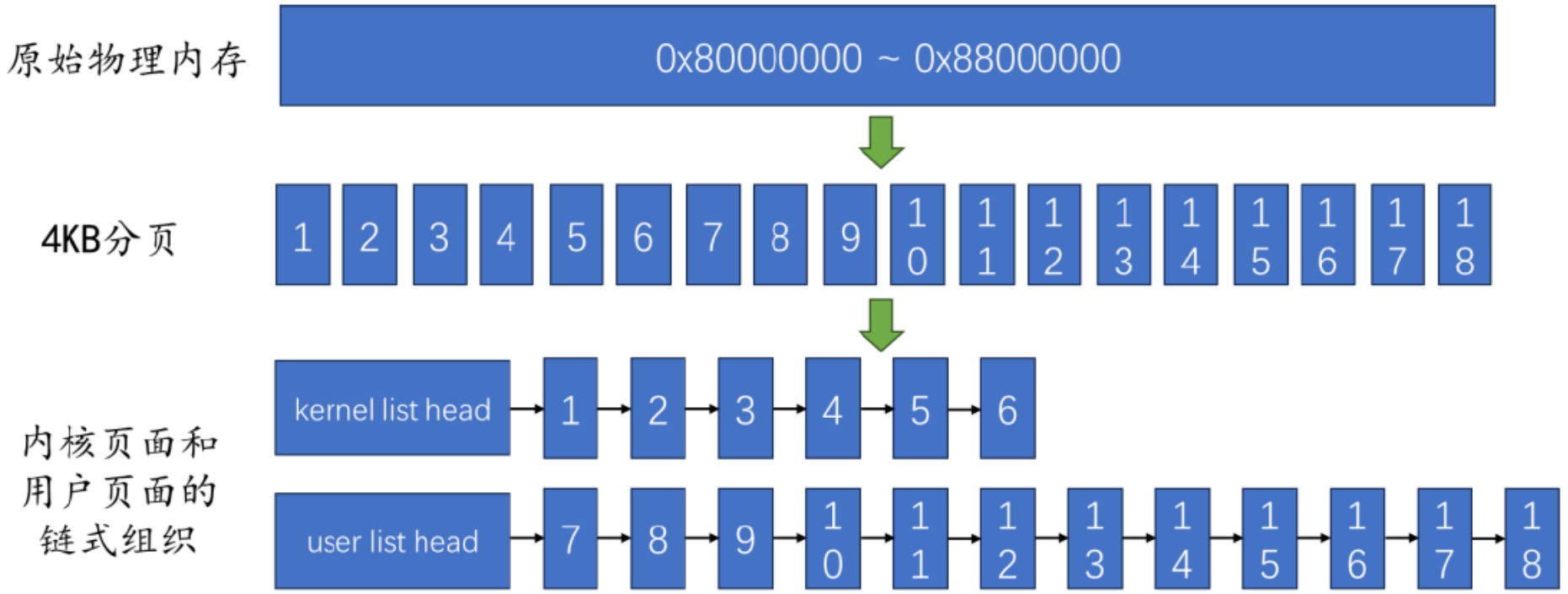
str.c（lib） pmen.c kvm.c

实验二：页表与内存管理

- 文件 **kernel.ld** 文件中规定了内核文件 **kernel-qemu** 在载入内存时的布局。
- 我们可以粗略把物理内存分成三个部分：
  - **0x80000000 ~ KERNEL_DATA** 存放了 **内核代码**
  - **KERNEL_DATA ~ ALLOC_BEGIN** 存放了 **内核数据**
  - **ALLOC_BEGIN ~ ALLOC_END** 属于 **未使用可分配的物理页**
- 只实现三个函数
  - void pmem_init(); // 初始系统, 只调用一次
  - void* pmem_alloc(); // 申请可用的物理页
  - void pmem_free(); // 释放申请了的物理页

实验二：页表与内存管理

- 物理内存管理方式



实验二：页表与内存管理

- 在pmen.c中定义

```
// 物理页节点
 typedef struct page_node {
struct page_node* next;
 } page_node_t;

// 许多物理页构成一个可分配的区域
typedef struct alloc_region {
uint64 begin; // 起始物理地址
 uint64 end; // 终止物理地址
 spinlock_t lk; // 自旋锁(保护下面两个变量)
 uint32 allocable; // 可分配页面数
 page_node_t list_head; // 可分配链的链头节点
} alloc_region_t;
// 内核和用户可分配的物理页分开
 static alloc_region_t kern_region, user_region;
```

在每个 **allocable page(4096 Byte)**的起始 **8 Byte** 存储一个指针，它指向下一个 **page**

只需要给一个固定的链表头list_head构成这样的结构：

list_head->page A->page B->page C->.....-> NULL

那么alloc操作就是移除：list_head->next；

free操作：将page插入list_head->next(头插法)

实验二：页表与内存管理

- 内核和用户都会申请和使用物理页，如果有一个恶意的用户程序，不断申请物理页且从不释放那么当内核需要申请物理页时就会发现物理页耗尽了，内核被用户攻击卡死是不可接受的。
- 需要将内核物理页和用户物理页分开，定义KERNEL_PAGES作为内核占用的物理页数目。
- 于是 ALLOC_BEGIN ~ ALLOC_END 被分成两部分
  - ALLOC_BEGIN ~ ALLOC_BEGIN + KERNEL_PAGES * PGSIZE 内核使用
  - ALLOC_BEGIN + KERNEL_PAGES * PGSIZE ~ ALLOC_END 用户使用

使用kern_region，user_region分别标识，alloc和free操作需要注明是否in_kernel。

实验二：页表与内存管理

内核态虚拟内存管理

- 考虑如何管理这个数据结构，也就是非常经典的 vm_mappages。

```
typedef struct {
    uint64 pa; // 物理地址
} addr_t;
addr_t pgtbl[VA_NUM];
```

- 开一个巨长的数组，数组长度等于所有虚拟页的数量（比如支持4GB虚拟内存，就是4GB/4KB）。
  这样就能用O(1)的复杂度快速查询了，但是代价是巨大的空间浪费。

- 需要：空间和时间的折中
  - 由于巨大的虚拟内存空间很难被完全使用，有时候只是使用最高处和最低处两部分，所以可以使用树形的三级映射来节约空间，同时时间开销也是O(1)级别的，虽然比不上数组法的O(1)。

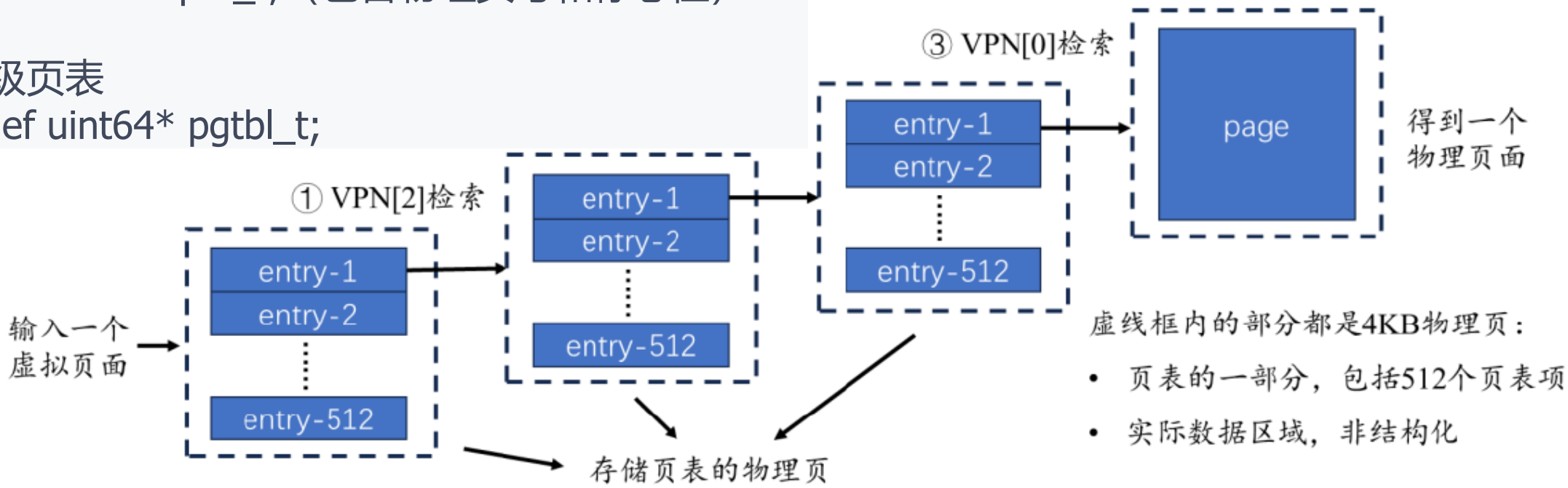
实验二：页表与内存管理

- 内核态虚拟内存管理
  - 在开始这部分代码的编写前，应该先阅读 **vmem.h** 里面的注释；
  - 注释里介绍了虚拟内存的一个规范 **SV39** 即 39 bit 虚拟地址的虚拟内存

实验二：页表与内存管理

```
// 页表项
 typedef uint64 pte_t;（包含物理页号和标志位）

// 顶级页表
typedef uint64* pgtbl_t;
```



页表本身也是存储在物理页中，我们称这种物理页叫页表物理页，特点是它的 PTE_R PTE_W PTE_X 都是0。

实验二：页表与内存管理

- 地址映射：对页表中某些页表项的修改（或增删）大部分是设置它存储的物理页号，有时也会设置或修改它的标志位(比如从 PTE_R 改成 PTE_R | PTE_W)。
- **vmem.c**中函数实现顺序如下：

  vm_getpte -> vm_mappages -> vm_unmappages
  - void vm_print(pgtbl_t pgtbl);
  - pte_t* vm_getpte(pgtbl_t pgtbl, uint64 va, bool alloc);
  - void vm_mappages(pgtbl_t pgtbl, uint64 va, uint64 pa, uint64 len, int perm);
  - void vm_unmappages(pgtbl_t pgtbl, uint64 va, uint64 len, bool freeit);

实验二：页表与内存管理

- 内核页表 kernel_pgtbl
  - kvm_init -> kvm_inithart
  - kernel_pgtbl 的映射大致可以划分成两部分：
  - ✓ 一部分是硬件寄存器区域，这部分的物理地址不是我们可以使用的，可以理解为QEMU保留的"假地址"
  - ✓ 另一部分是可用内存区域，即0x80000000 到 0x80000000 + 128 MB，内核页表对这两部分的映射都是虚拟地址等于物理地址的直接映射。

映射完毕后kernel_pgtbl就可以上线工作了，把它写入satp寄存器即可正式开启MMU翻译。

终于结束了直接访问物理地址的时代，此后的所有地址访问，只要satp寄存器里不是0，本质都是访问虚拟地址，因为 MMU 会忠实地将所有地址访问 va 经过页表翻译变成 pa。