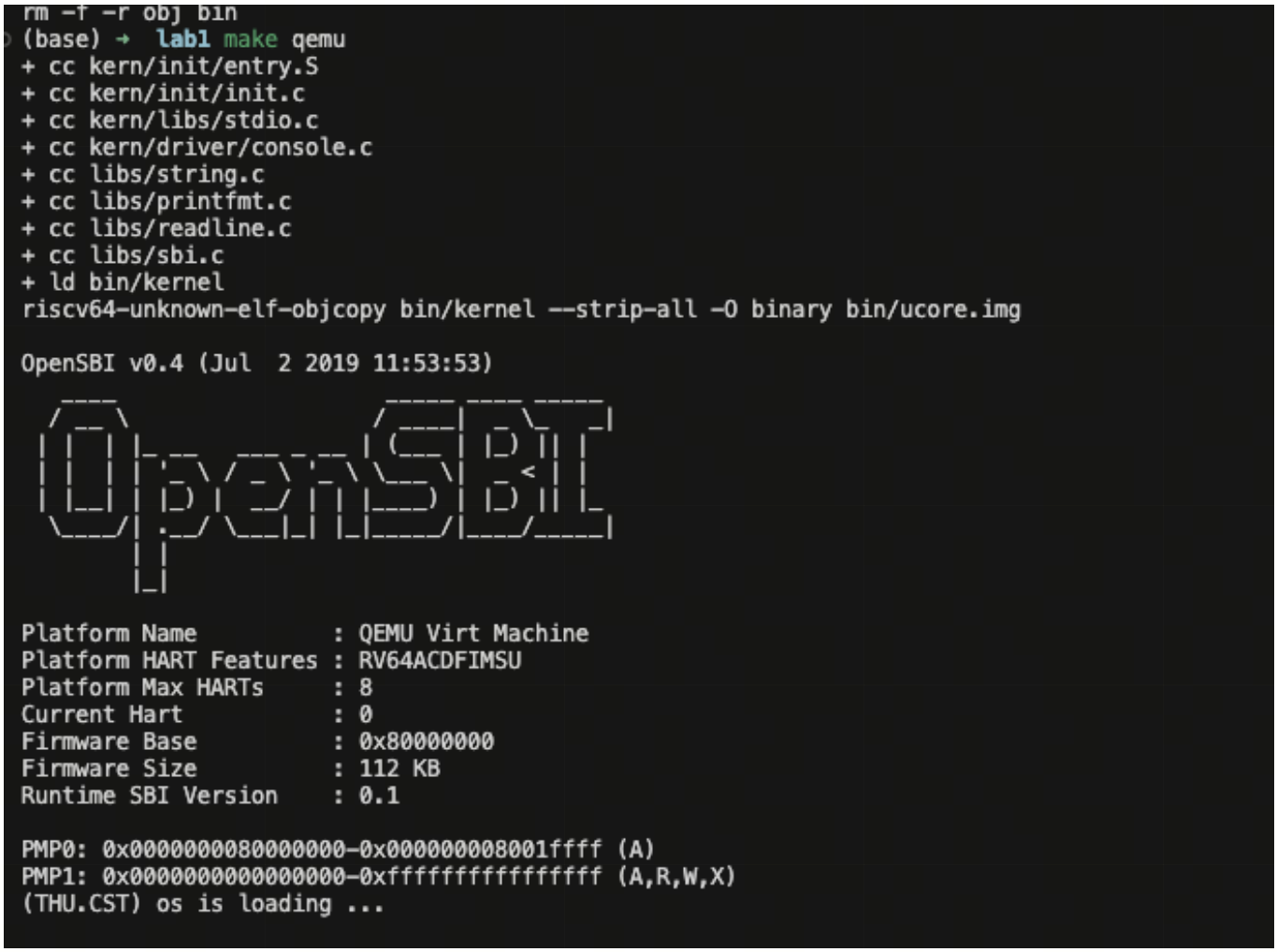
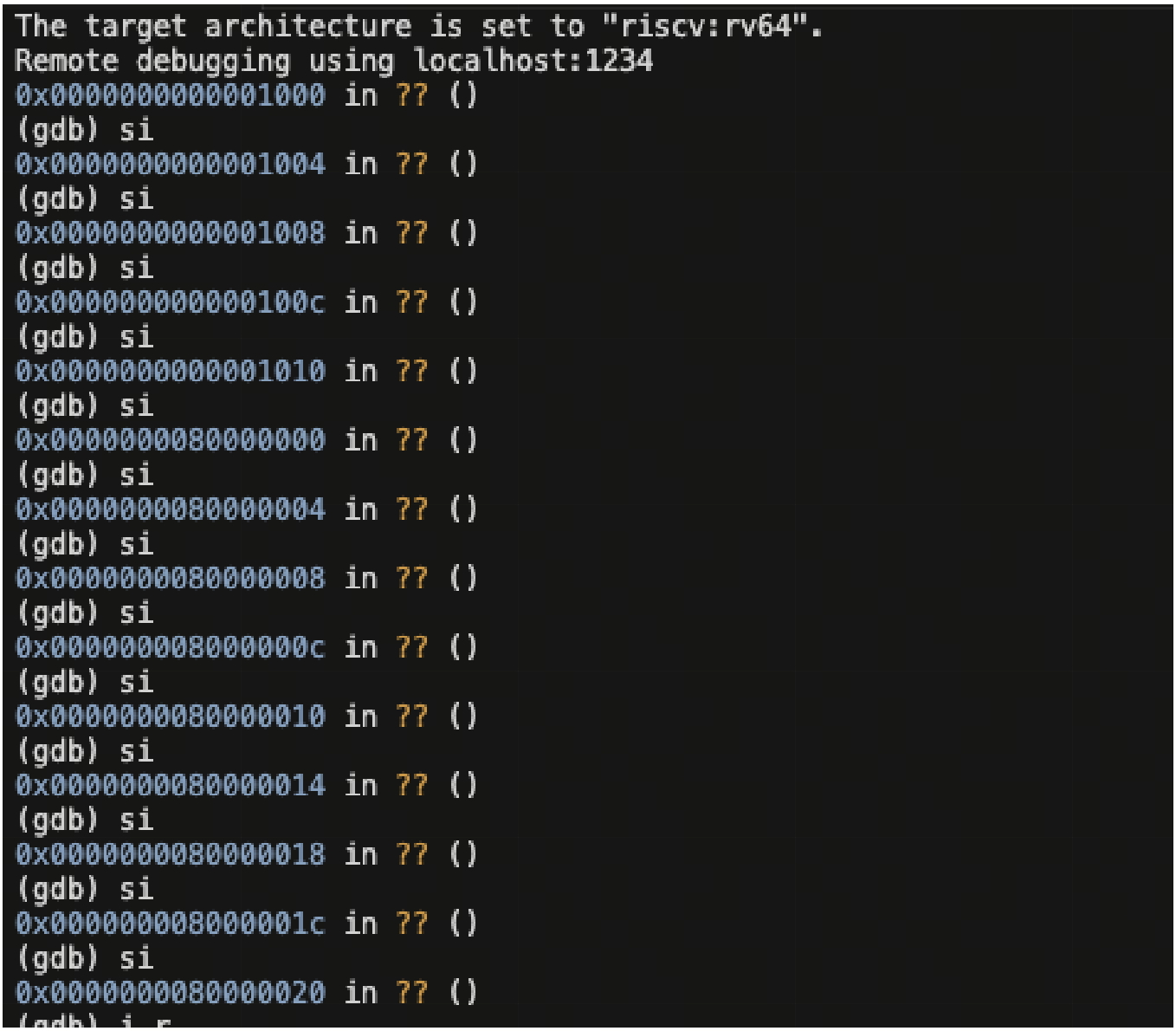
OS

启动

首先，通过make qemu得到如下状态，说明我们的makefile是正确的:



接着我们使用如下命令make debug+make gdb进入调试状态:



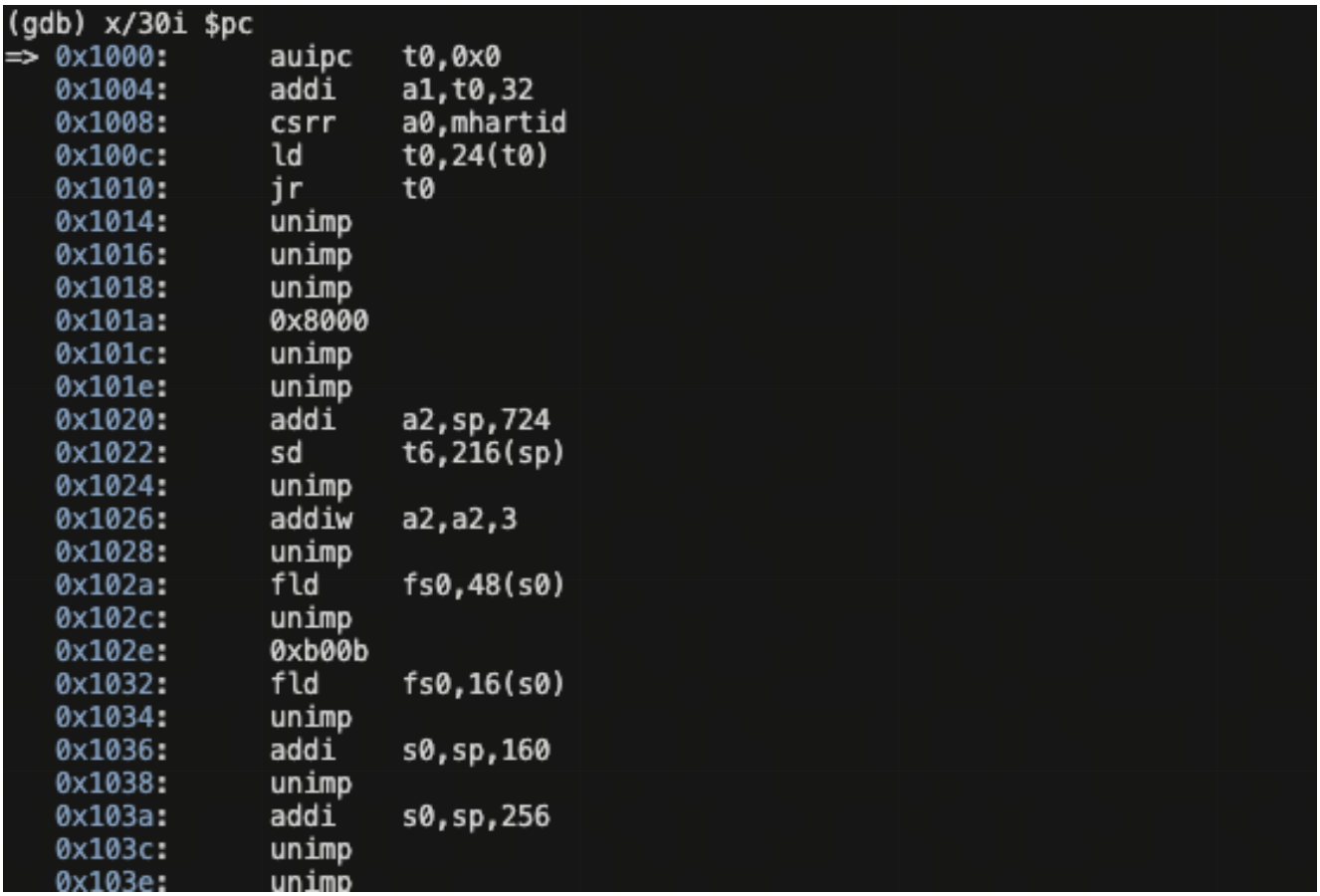
我们发现程序起始的地址是0x1000，这时候打印一下寄存器的值如下

```
(gdb) i r
ra             0x0        0x0
sp             0x0        0x0
gp             0x0        0x0
tp             0x0        0x0
t0             0x8000001c       2147483676
t1             0x80000000       2147483648
t2             0x0        0
fp             0x0        0x0
s1             0x0        0
a0             0x0        0
a1             0x1020     4128
a2             0x0        0
a3             0x0        0
a4             0x0        0
a5             0x0        0
a6             0x0        0
a7             0x0        0
s2             0x0        0
s3             0x0        0
s4             0x0        0
s5             0x0        0
s6             0x0        0
s7             0x0        0
s8             0x0        0
s9             0x0        0
s10            0x0        0
s11            0x0        0
t3             0x0        0
t4             0x0        0
t5             0x0        0
t6             0x0        0
pc             0x80000020       0x80000020
dscratch       Could not fetch register "dscratch"; remote failure reply 'E14'
mucounteren    Could not fetch register "mucounteren"; remote failure reply 'E14'
```

1. CPU 从复位地址（`0x1000`）开始执行初始化固件（OpenSBI）的汇编代码，进行最基础的硬件初始化。

> 💡 OpenSBI（Open Source Supervisor Binary Interface）是 RISC-V 架构下运行在机器模式（M 模式）的一层开源固件，用于在硬件与操作系统内核（如 Linux、FreeBSD）之间提供标准化的接口和初始化支持。它负责完成 CPU 核心初始化、中断与定时器配置、内存区域管理、异常委托、SBI 调用（如打印、关机、核间通信）等底层任务，相当于 RISC-V 平台上的“BIOS + 固件 + Supervisor API” 组合。操作系统内核运行在监督模式（S 模式）时，会通过 SBI 调用访问这些底层服务，例如通过 `sbi_call` 请求 OpenSBI 打印字符或设置时钟中断。简而言之，OpenSBI 是 RISC-V 平台从上电到启动 OS 之间的关键桥梁，负责完成环境初始化和特权级切换，为更上层的软件（如 Linux 内核）提供统一的底层抽象

通过make gdb 输入x/30i $pc 查看即将执行的30条汇编指令，其中在地址为 0x1010的指令处会跳转，故实际执行的为以下指令：



这段汇编代码的作用主要是复制t0寄存器 然后跳转到地址0x80000000，以下是涉及到指令的简单介绍：

📍 `auipc`（Add Upper Immediate to PC）是RISC-V中的U型指令，用于将一个20位立即数左移12位后与当前指令地址（PC）相加，并将结果写入目标寄存器 `rd`，其操作为 `rd = PC + (imm << 12)`，常用于生成PC相对地址（如跳转、访问全局变量等），区别于 `lui`（基于0构造常量）， `auipc` 是基于当前PC构造地址的。

`csrr` 是 RISC-V 中用于从控制与状态寄存器（CSR, Control and Status Register）读取值的指令，全称为 *Control and Status Register Read*。其操作是将指定 CSR 的内容读入通用寄存器 `rd`，并可在部分变体（如 `csrrw`，`csrrs`，`csrrc`）中同时对 CSR 执行写、置位、清位等操作。基本形式为 `csrr rd, csr`，功能为 `rd = CSR[csr]`，主要用于读取如 `mstatus`、`medeleg`、`mtvec`、`time` 等机器级寄存器，用于操作系统或异常控制逻辑中。

① **CSR（Control and Status Register）**：控制与状态寄存器，是 RISC-V 中用于管理特权级、异常、时间中断等系统控制信息的特殊寄存器集合；

② **rd**：目标寄存器（destination register），用于保存指令执行结果的通用寄存器；

③ **csrrw / csrrs / csrrc**：分别表示 CSR 读写（Write）、读置位（Set）、读清位（Clear）的变体指令，可在读的同时修改 CSR；

④ **mstatus**：机器状态寄存器，存放全局中断使能和特权级状态；

⑤ **medeleg**：异常委托寄存器，用于将异常从机器模式委托到监督模式；⑥ **mtvec**：机器陷阱向量基址寄存器，指定异常或中断处理函数的入口地址；

⑦ **time**：机器定时器寄存器，记录当前计时器的时间值，用于时钟中断与系统调度。

图中 `mhartid` 是 RISC-V 的机器级 CSR（控制与状态寄存器），表示当前**硬件线程（hart）的 ID**，在多核或多线程系统中用于区分不同的 CPU 内核，每个 hart 都有唯一的 `mhartid` 值；

`ld` 是 RISC-V 64 位架构中的加载指令，全称为 **Load Doubleword**，用于从内存中读取一个 64 位（8 字节）数据并存入目标寄存器。其基本格式为 `ld rd, offset(rs1)`，表示从地址 `rs1 + offset` 处取出 8 字节数据加载到寄存器 `rd` 中，`offset` 是 12 位有符号立即数，用于实现基址寻址。它通常用于从内存读取变量或栈数据，例如 `ld x5, 8(x10)` 表示将内存地址 `x10 + 8` 处的 64 位内容读入 `x5`。

`unimp` 是 RISC-V 汇编中的“未实现指令”（unimplemented instruction）伪操作，通常不是一条真实存在于指令集的机器指令，而是汇编器生成的一种占位指令，用来表示“此处功能尚未实现”或“保留位置”；在机器码层面，它一般会被编码为一个非法操作码，从而在执行时触发 **illegal instruction trap（非法指令异常）**，常用于调试、占位或提示未实现功能（类似于 C 语言中的 `assert(0)` 或 `TODO` 标记）。

```
(gdb) watch t0
No symbol "t0" in current context.
(gdb) watch $t0
Watchpoint 1: $t0
(gdb) si
0x0000000000001008 in ?? ()
(gdb) si
0x000000000000100c in ?? ()
(gdb) si

Watchpoint 1: $t0

Old value = 4096
New value = 2147483648
0x0000000000001010 in ?? ()
(gdb)
```

我们可以发现t0寄存器的值变为了要跳转的地址 使用x/4gx $t0+24可以查看ld t0,24(t0)指令的结果，即内存[t0 + 24]处的值为0x80000000,这个值被赋值给t0

```
(gdb) si
0x0000000000001008 in ?? ()
(gdb) x/4gx $t0+24
0x1018: 0x0000000080000000      0x260d0000edfe0dd0
0x1028: 0xb00b000038000000      0x1100000028000000
```

2. SBI 固件进行主初始化，其核心任务之一是将内核加载到 `0x80200000` 。可以使用 `watch *0x80200000` 观察内核加载瞬间，避免单步跟踪大量代码

> 🏆 该地址处加载的是作为bootloader的 OpenSBI.bin，该处的作用为加载操作系统内核并启动操作系
>
> 统的执行。

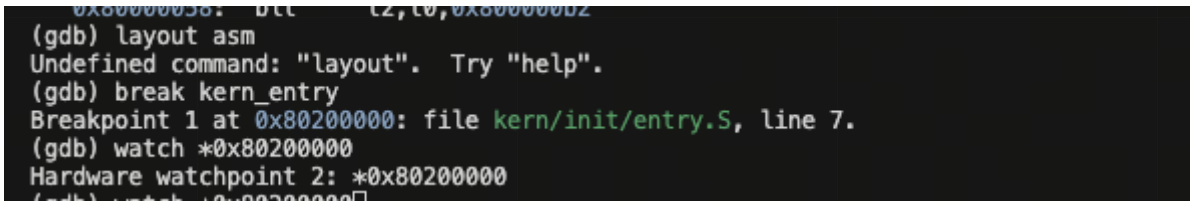
```
(gdb) si
0x0000000080000000 in ?? ()
(gdb) x/24i 0x80000000
=> 0x80000000:  csrr    a6,mhartid
   0x80000004:  bgtz    a6,0x80000108
   0x80000008:  auipc   t0,0x0
   0x8000000c:  addi    t0,t0,1032
   0x80000010:  auipc   t1,0x0
   0x80000014:  addi    t1,t1,-16
   0x80000018:  sd      t1,0(t0)
   0x8000001c:  auipc   t0,0x0
   0x80000020:  addi    t0,t0,1020
   0x80000024:  ld      t0,0(t0)
   0x80000028:  auipc   t1,0x0
   0x8000002c:  addi    t1,t1,1016
   0x80000030:  ld      t1,0(t1)
   0x80000034:  auipc   t2,0x0
   0x80000038:  addi    t2,t2,988
   0x8000003c:  ld      t2,0(t2)
   0x80000040:  sub     t3,t1,t0
   0x80000044:  add     t3,t3,t2
   0x80000046:  beq     t0,t2,0x8000014e
   0x8000004a:  auipc   t4,0x0
   0x8000004e:  addi    t4,t4,260
   0x80000052:  sub     t4,t4,t2
   0x80000056:  add     t4,t4,t0
   0x80000058:  blt     t2,t0,0x800000b2
```

这段汇编代码的含义如下：

1. `csrr a6, mhartid`：读取机器级 CSR `mhartid`（硬件线程 ID）到寄存器 `a6`，用于判断当前是哪个核在运行。
2. `bgtz a6, 0x80000108`：若 `a6` > 0（即非主核），跳转到 `0x80000108`；说明主核负责初始化，其余核等待。
3. `auipc t0, 0x0` / `addi t0, t0, 1032`：组合出一个地址（`t0 = PC + 1032`），通常用于定位某个表或数据段。

4. `auipc t1, 0x0` / `addi t1, t1, -16`：同样构造出相对当前 PC 的地址，可能用于读取或写入初始化数据。
5. `sd t1, 0(t0)`：将 `t1` 的值（一个地址）写入以 `t0` 为基址的内存，可能是初始化表项。
6. `auipc t0, 0x0` / `addi t0, t0, 1020` / `ld t0, 0(t0)`：取出内存中存放的指针或函数地址，准备跳转或比较。
7. `auipc t1, 0x0` / `addi t1, t1, 1016` / `sub t3, t1, t0`：计算两个地址差，常用于算偏移或循环界限。
8. `auipc t2, 0x0` / `addi t2, t2, 988` / `sub t3, t1, t2`：继续构造并比较地址范围。
9. `beq t0, t2, 0x8000014e`：若 `t0 == t2`，跳转至 `0x8000014e`，可能是结束或下一阶段初始化入口。
10. `auipc t4, 0x0` / `addi t4, t4, 260` / `sub t4, t4, t2` / `add t4, t4, t0`：构造另一个偏移地址，用于计算加载或拷贝目标位置。
11. `blt t2, t0, 0x800000b2`：若 `t2 < t0`，跳回 `0x800000b2`，形成循环结构，说明这段逻辑可能在遍历或复制一段内存区域（如初始化 `.bss` 或 `.data` 段）

接着输入指令 break kern_entry，在目标函数kern_entry的第一条指令处设置断点，输出如下：



这个文件的主要作用是初始化内核的基本环境（如栈），然后跳转到 C 语言的初始化函数 kern_init，从而启动整个内核初始化流程（包括清零 BSS 段、打印启动信息等）。

代码块

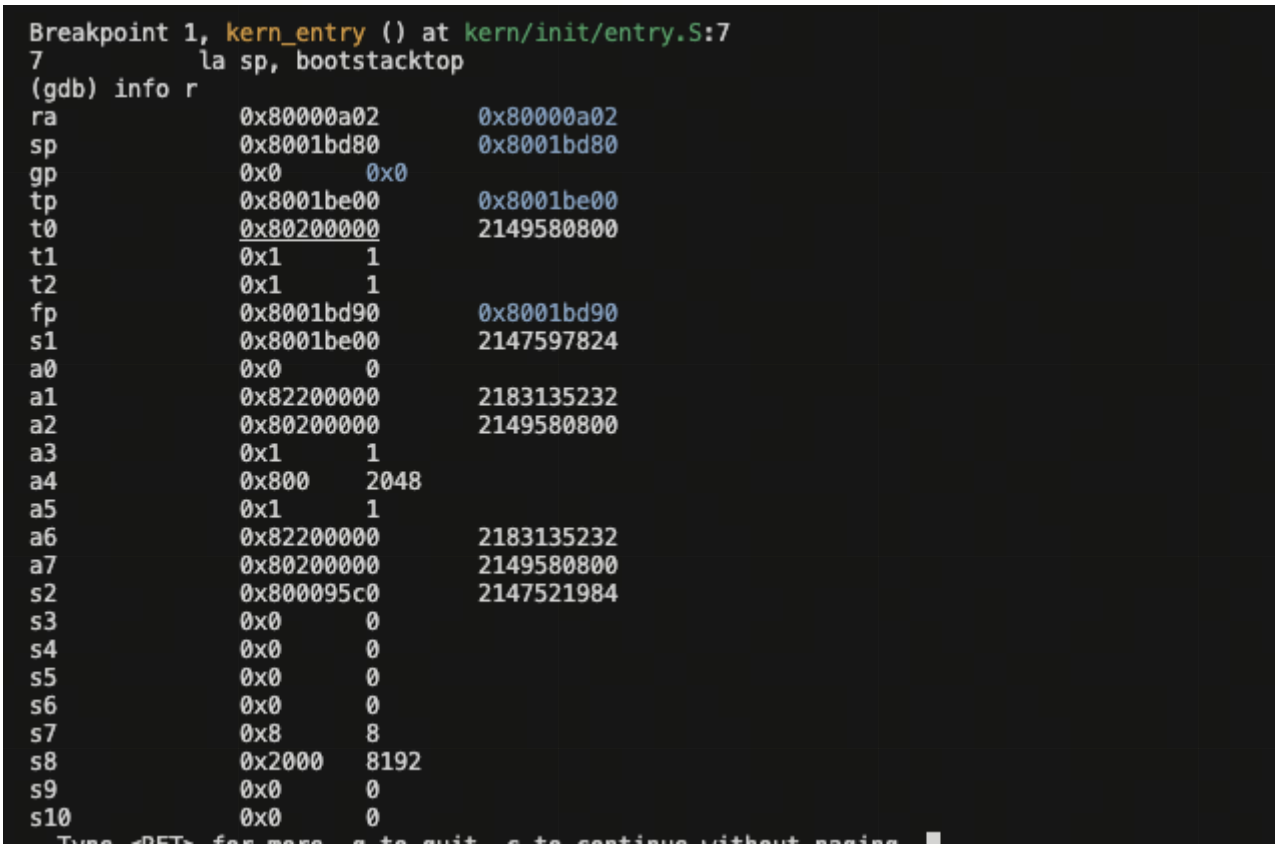
```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
```

```
16      .space KSTACKSIZE
17      .global bootstacktop
18  bootstacktop:
```



la 是 RISC-V 汇编的伪指令（pseudo-instruction），全称为 "load address"，它将符号 bootstacktop 的绝对地址加载到寄存器 sp（栈指针寄存器，stack pointer）中。具体来说：

- bootstacktop 是一个全局符号，定义在同一个文件的 .data 段中（第 18 行：.global bootstacktop），它指向内核引导栈（bootstack）的顶部地址。
- 这个栈是一个预分配的内核栈空间（第 15-18 行：从 bootstack 开始，分配 KSTACKSIZE 大小的空间，即 2 个页，即 8KB），位于内核的低地址区（由 memlayout.h 中的 KSTACKSIZE 定义）。
- 执行后，sp 寄存器被设置为栈顶地址（高地址），栈从高向低生长（RISC-V 的栈生长方向）。
- 目的：

在操作系统内核启动流程中，内核被加载到内存后，CPU 的寄存器状态不确定，尤其是栈指针 sp 可能未初始化或指向无效位置。如果不设置栈，任何函数调用（包括参数传递、局部变量分配、返回地址保存）都会失败，导致内核崩溃。这个指令的目的是初始化内核的引导栈（boot stack），为后续的 C 语言函数调用提供一个可靠的栈空间。内核启动早期（如调用 kern_init）需要栈来处理中断、函数调用和局部变量。在 ucore 中，这个引导栈是临时的内

核栈，后续会切换到每个进程的独立栈。结合流程：在 bootloader 跳转到 kern_entry 后，这是第一个操作，确保内核有栈可用，然后才能安全地跳转到初始化代码。

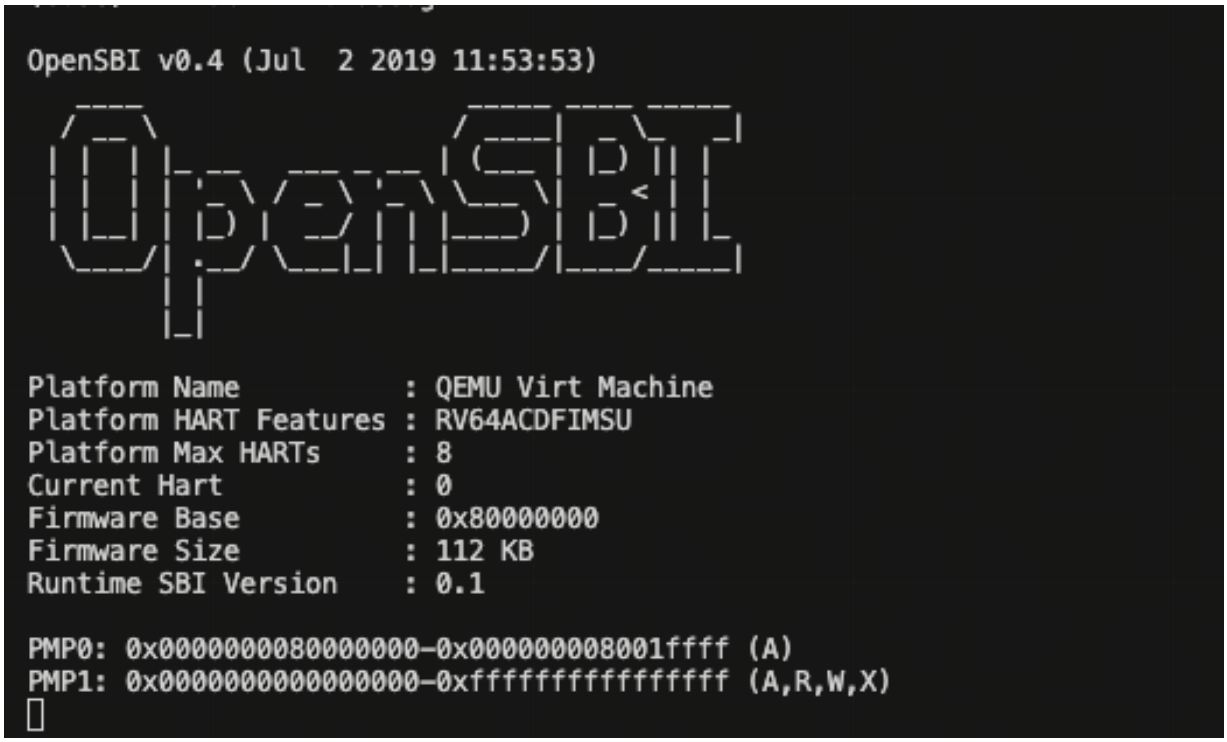
tail 是 RISC-V GCC 的伪指令（类似于 jalr，jump and link register），它生成一条无条件跳转指令：jalr zero, kern_init（跳转到 kern_init 的地址，并将返回地址保存到 zero 寄存器）。

- zero 寄存器是 RISC-V 的常量寄存器（始终为 0），所以不会保存返回地址，这相当于一个尾调用（tail call），直接跳转而不返回。
- kern_init 是 C 语言函数（定义在 kern/init/init.c 中），它标记为 attribute((noreturn))，表示函数不会返回（无限循环结束）。
- 执行后，控制流直接转移到 kern_init 的入口，kern_entry 后的代码不会再执行。
- 目的：

在内核启动流程中，kern_entry 是汇编入口，用于处理架构特定的初始化（如设置栈），而实际的内核逻辑（如清零数据段、初始化设备、打印消息）在 C 代码中实现。这个指令的目的是将控制权无条件转移到内核初始化函数 kern_init，开始执行内核的启动序列，而不返回到入口点（因为入口点已完成其唯一任务）。这样设计避免了不必要的返回栈帧开销（tail call 优化），并确保内核启动是线性的：从 bootloader → kern_entry（设置栈）→ kern_init（初始化 BSS 段、调用 cprintf 打印 "(THU.CST) os is loading ..."、进入无限循环）。在 ucore 中，kern_init 是内核 bootstrapping 的起点，后续会扩展到 trap 初始化、内存管理等，但 lab1 阶段仅到打印消息。如果不跳转，内核会卡在入口点；这个 tail call 确保流程继续到 C 层面的初始化。

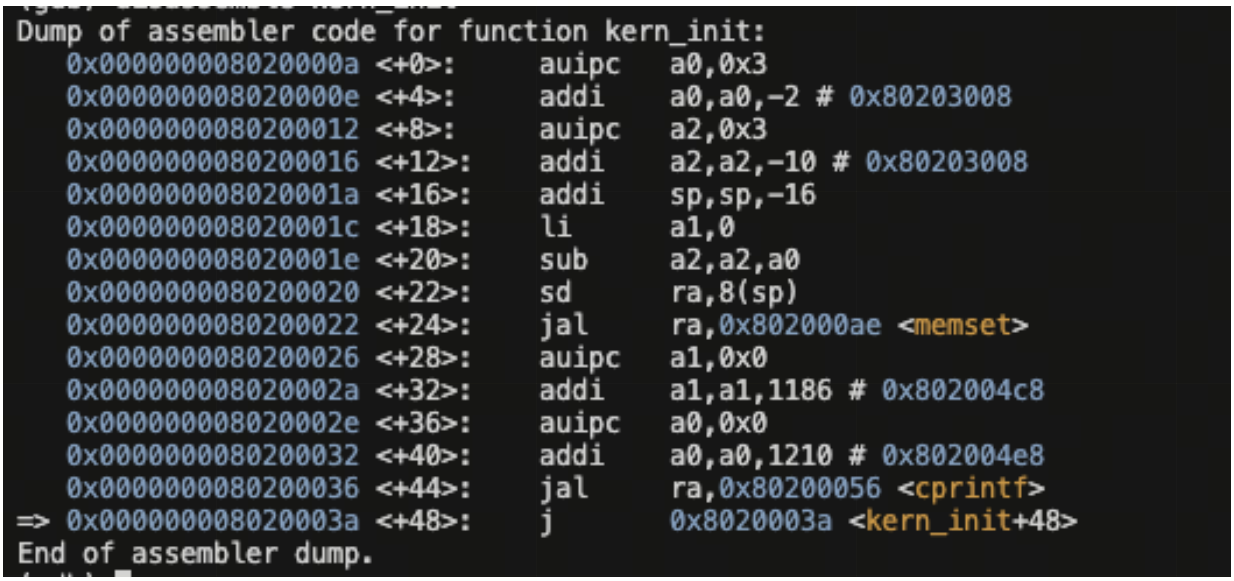
输入x/10i 0x80200000,地址 0x80200000由 kernel.ld中定义的 BASE_ADDRESS（加载地址）所决定，标签 kern_entry是在 kernel.ld中定义的 ENTRY（入口点）

可以看到在 kern_entry之后，紧接着就是 kern_init，同时debug输出如下



3. SBI 最后跳转到 0x80200000，将控制权移交内核。使用 `b *0x80200000` 可在此中断，验证内核开始执行

接着输入指令 break kern_init



可以看到这个函数最后一个指令是 j 0x8020003c <kern_init+48>，也就是跳转到自己，所以代码会在这里一直循环下去。

物理内存和页表

Best-fit

参考first-fit只需要更改找到目标页的逻辑即可

代码块

```c
static struct Page *
best_fit_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;
    size_t min_size = nr_free + 1;
    while ((le = list_next(le)) != &free_list) {
        struct Page *p = le2page(le, page_link);
        if (p->property >= n && p->property < min_size) {
            min_size = p->property;
            page = p;
        }
    }

    if (page != NULL) {
        list_entry_t* prev = list_prev(&(page->page_link));
        list_del(&(page->page_link));
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add(prev, &(p->page_link));
        }
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

buddy-sys

全局结构与不变量

我们把所有空闲物理页按阶（order）管理：阶 `i` 的块大小为 `2^i` 页。每个阶对应一个双向链表 `buddy_area[i].free_list`，链表中挂的是**块头页**（head page），且只有块头页会被标记 `PageProperty=1`，并在其 `Page.property` 字段里记录该块的阶次 `i`。同一阶的一对伙伴块（buddies）在全局页数组 `pages[]` 的**索引**上互为 `idx ^ (1<<i)` 的关系。实现坚持以下不变量：

① 只有块头页被置位 `PageProperty` 且其 `property==阶`；

② 任一阶的空闲链表中**不能同时**出现两个互为伙伴关系的块头，否则说明未及时合并（实现中在插入时主动向上合并来避免）；

③ 按“以页为单位”维护计数：`buddy_area[i].nr_free` 记录该阶空闲页**总数**（不是块数），全局 `buddy_total_free` 是各阶空闲页数之和；

④ 释放或导入空闲内存时，先按“最大对齐”的贪心策略切分成若干 `2^k` 页块，再通过伙伴检测自底向上合并，以保持空闲空间尽量粗粒度、便于大块分配。

代码块

```
#define BUDDY_MAX_ORDER  18

typedef struct {
    list_entry_t free_list;
    unsigned int nr_free;
} buddy_area_t;

static buddy_area_t buddy_area[BUDDY_MAX_ORDER + 1];
static size_t buddy_total_free = 0;

#define order_free_list(order) (buddy_area[(order)].free_list)
#define order_nr_free(order)   (buddy_area[(order)].nr_free)
```

内联工具：`page_index / index_to_page / order_block_pages`

`static inline size_t page_index(struct Page *p)` 与 `static inline struct Page *index_to_page(size_t idx)` 是在全局页数组 `pages[]` 与“页索引”之间的双向转换。Buddy 算法的伙伴定位、块大小计算都依赖**索引级**的位运算，因而实现统一先把指针映射为索引做数学运算，再映射回指针。`static inline size_t order_block_pages(size_t order)` 返回某阶块包含的页数 `1u<<order`，用于计算伙伴偏移（`idx ^ (1<<order)`）、计数增减以及循环提升阶次。

代码块

```
static inline size_t page_index(struct Page *p) { return (size_t)(p - pages); }
static inline struct Page *index_to_page(size_t idx) { return pages + idx; }
static inline size_t order_block_pages(size_t order) { return ((size_t)1u << order); }
```

初始化管理器：`buddy_init(void)`

`buddy_init` 负责**结构层**初始化：遍历 `0..BUDDY_MAX_ORDER`，调用 `list_init(&order_free_list(i))` 把每阶空闲链表置空，把 `order_nr_free(i)` 清零；同时把 `buddy_total_free` 置 0。注意它并**不**向系统宣告任何可用页；

只是把管理结构复位为“空仓库”。真正把物理内存导入为可分配，是 `buddy_init_memmap` 的职责。

代码块

```
static void buddy_init(void) {
    for (unsigned int i = 0; i <= BUDDY_MAX_ORDER; i++) {
        list_init(&order_free_list(i));
        order_nr_free(i) = 0;
    }
    buddy_total_free = 0;
}
```

导入空闲区：`buddy_init_memmap(base, n)`

`buddy_init_memmap(base, n)` 把连续区 `[base, base+n)` 导入为可分配空闲页。首先对区间内每页做一致性与清理：断言 `PageReserved(p)`（说明这段页此前由内核保留、未被 buddy 管理），随后清 `flags`、清 `property`、把引用计数设 0；这保证导入的页没有残留“空闲块头”标记或使用中状态。然后调用 `insert_region(base, n)`：它会按“最大可对齐块”的贪心方式把区间切分为若干块并在插入时向上合并，确保导入后空闲链表处于“无可进一步合并”的规范状态。最后把全局空闲总页 `buddy_total_free += n`。

代码块

```
static void buddy_init_memmap(struct Page *base, size_t n) {
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p++) {
        assert(PageReserved(p));
        p->flags = 0;
        p->property = 0;
        set_page_ref(p, 0);
    }
    insert_region(base, n);
    buddy_total_free += n;
}

```

关键插入+合并：`try_coalesce_and_insert(base, order)`

`try_coalesce_and_insert` 的角色是：给定一个“当前被视为阶 `order` 的空闲块头 `base`”，尽可能把它与同阶伙伴自底向上合并，然后把得到的更大块挂回对应阶链表。它先取 `idx = page_index(base)`，计算伙伴索引 `buddy_idx = idx ^ order_block_pages(order)`，并取伙伴头页 `buddy_head = index_to_page(buddy_idx)`。只有在**两个条件同时成立**时才可合并：

① `PageProperty(buddy_head)` 为真，说明伙伴确实是空闲块头；

② `buddy_head->property == order`，说明伙伴和当前块处于同一阶。

若可合并，则把伙伴从该阶链表里摘除（`list_del`），把该阶空闲页计数减去**一个块的页数**（`nr_free[order] -= 1<<order`），清除伙伴的块头标记（`ClearPageProperty` 且 `property=0`），并把合并后新块的块头规范化为**较小索引**的一端（如果 `buddy_idx < idx` 则 `base=buddy_head; idx=buddy_idx`）。随后 `order++`，继续尝试与更高阶的伙伴合并。循环结束（遇到伙伴不存在/非同阶/不是空闲块头，或已达 `BUDDY_MAX_ORDER`）后，将最终块 `base` 标记为空闲块头（设 `base->property=order; SetPageProperty(base)`），将其插入该阶链表（`list_add`），并把该阶空闲页计数**增加一个完整块的页数**。这一过程确保每次插入后的链表没有可进一步合并的伙伴，从而维持不变量 ②。

代码块

```
static void try_coalesce_and_insert(struct Page *base, unsigned int order) {
    size_t idx = page_index(base);
    while (order < BUDDY_MAX_ORDER) {
        size_t buddy_idx = idx ^ order_block_pages(order);
        struct Page *buddy_head = index_to_page(buddy_idx);
        if (PageProperty(buddy_head) && buddy_head->property == order) {
            list_del(&(buddy_head->page_link));
            order_nr_free(order) -= order_block_pages(order);
            ClearPageProperty(buddy_head);
            buddy_head->property = 0;
            if (buddy_idx < idx) {
                base = buddy_head;
                idx = buddy_idx;
            }
            order++;
            continue;
        }
        break;
    }
    base->property = order;
    SetPageProperty(base);
    list_add(&order_free_list(order), &(base->page_link));
    order_nr_free(order) += order_block_pages(order);
}
```

区间插入：`insert_region(base, n)`

`insert_region` 把一个**连续的**空闲区 `[base, base+n)` 切成若干尽可能大的、对齐的 power-of-two 块，然后逐块调用 `try_coalesce_and_insert`。它维护当前指针的全局索引 `idx = page_index(base)` 与剩余页数 `n`，在每轮中寻找最大的阶 `order` 使得：

① `2^(order+1) <= n`（块大小不超过剩余），

② 当前索引 `idx` 按该大小对齐（`(idx & (blk-1)) == 0`）。

用这种“最大对齐优先”的策略，可以保证切出来的每块都是**天然与其伙伴对齐**的候选，这样传给 `try_coalesce_and_insert` 时，就能顺利触发**向上合并**，把相邻的同阶块（如果存在）合成更大的块，最终以尽量大的块粒度挂回空闲链表。此外，这种贪心法避免了产生跨阶的小碎块，是 Buddy 系统导入或归还大段内存的标准做法。每次插入后把 `idx += 块大小`，`n -= 块大小`，直到区间处理完毕。

代码块

```
1
2   static void insert_region(struct Page *base, size_t n) {
3       size_t idx = page_index(base);
4       while (n > 0) {
5           unsigned int order = 0;
6           size_t max_fit = 1;
7           while (order < BUDDY_MAX_ORDER) {
8               size_t blk = order_block_pages(order + 1);
9               if (blk > n) break;
10              if ((idx & (blk - 1)) != 0) break;
11              order++;
12              max_fit = blk;
13          }
14          try_coalesce_and_insert(index_to_page(idx), order);
15          idx += max_fit;
16          n   -= max_fit;
17      }
18  }
```

后缀回收：`carve_suffix_as_free(base, n, s)`

`carve_suffix_as_free` 服务于“先取大块再返还未用部分”的分配策略：若实际分配需求为 `n` 页，而我们从高阶空闲链表中取到一个 `s` 页的大块（`s >= n`），那么返回区间 `[base, base+n)` 之外的**后缀** `[base+n, base+s)` 应被归还给 Buddy 系统。函数检测 `n < s` 时，把后缀起点 `suffix = base + n` 和后缀长度 `suffix_pages = s - n` 交给 `insert_region(suffix, suffix_pages)`。这样返还的后缀会被自动切成最大对齐块并尽可能与邻居合并，恢复到规范形态。

代码块

```
1   static void carve_suffix_as_free(struct Page *base, size_t n, size_t s) {
2       if (n < s) {
3           struct Page *suffix = base + n;
4           size_t suffix_pages = s - n;
5           insert_region(suffix, suffix_pages);
6       }
```

```
7   }
```

分配路径：`buddy_alloc_pages(size_t n)`

`buddy_alloc_pages` 执行“按页数 `n` 连续分配”。它首先断言 `n>0`，并检查**全局可用页**是否足够（`n > buddy_total_free` 直接失败返回 `NULL`，避免空跑）。接着计算能容纳 `n` 的最小阶 `want_order`（循环增加直到 `1<<want_order >= n` 或达上限），再从 `want_order` 向上扫描找到**第一个非空**的阶 `found_order`（First-Fit/近似 Best-Fit 的折衷，既尽量不浪费，又避免过度扫描低阶）。如果一直到 `BUDDY_MAX_ORDER` 都为空，则返回失败。找到后，从对应阶链表取出**一整块**：`list_next` 取一个块头，`list_del` 摘除，按整块页数从该阶的页计数里减去（注意这里按“页”而非“块数”减），清掉块头标志与阶标记（该块即将被“切分占用”，不应再被视为空闲块头）。现在手里有一个连续 `s=1<<found_order` 页的大区间；函数把**前 `n` 页**作为“分配结果”，剩余后缀 `[base+n, base+s)` 交由 `carve_suffix_as_free` 返还系统。为了避免返还之后，前 `n` 页误被合并回去，函数还遍历这 `n` 页逐一清 `PageProperty` / `property`（确保它们不被视为空闲块头）。最后把 `buddy_total_free -= n`，并返回分配区间的首页指针。

代码块

```
1   static struct Page *buddy_alloc_pages(size_t n) {
2       assert(n > 0);
3       if (n > buddy_total_free) {
4           return NULL;
5       }
6       // Find minimal order with block size >= n and non-empty free list (or
    larger order to carve)
7       unsigned int want_order = 0;
8       while ((order_block_pages(want_order) < n) && want_order <
    BUDDY_MAX_ORDER) {
9           want_order++;
10      }
11      unsigned int found_order = want_order;
12      while (found_order <= BUDDY_MAX_ORDER &&
    list_empty(&order_free_list(found_order))) {
13          found_order++;
14      }
15      if (found_order > BUDDY_MAX_ORDER) {
16          return NULL;
17      }
18
19      // Pop one block from found_order
20      list_entry_t *le = list_next(&order_free_list(found_order));
21      struct Page *block = le2page(le, page_link);
22      list_del(&(block->page_link));
23      order_nr_free(found_order) -= order_block_pages(found_order);
24      ClearPageProperty(block);
```

```
25      block->property = 0;
26
27      size_t s = order_block_pages(found_order);
28      carve_suffix_as_free(block, n, s);
29
30      struct Page *p = block;
31      for (size_t i = 0; i < n; i++, p++) {
32          ClearPageProperty(p);
33          p->property = 0;
34      }
35
36      buddy_total_free -= n;
37      return block;
38  }
```

释放路径：`buddy_free_pages(struct Page *base, size_t n)`

`buddy_free_pages` 释放连续 `n` 页。它先对要释放的每一页断言“不处于保留或仍被标‘空闲块头’的状态”（`!PageReserved(p) && !PageProperty(p)`），这是防御性编程，避免把**尚在空闲链表里**或**受保护**的页重复释放；随后清理每页 `flags` 与引用计数。接着直接调用 `insert_region(base, n)`：这一步把整段按最大对齐块切分并尝试与相邻伙伴**自底向上**反复合并，直到无法再合并为止，最后以规范形态挂回各阶链表。释放完成后，把 `buddy_total_free += n`。

代码块

```
1   static void buddy_free_pages(struct Page *base, size_t n) {
2       assert(n > 0);
3       struct Page *p = base;
4       for (; p != base + n; p++) {
5           assert(!PageReserved(p) && !PageProperty(p));
6           p->flags = 0;
7           set_page_ref(p, 0);
8       }
9       insert_region(base, n);
10      buddy_total_free += n;
11  }
```

查询空闲：`buddy_nr_free_pages(void)`

代码块

```
1   static size_t buddy_nr_free_pages(void) { return buddy_total_free; }
```

最小自检：`basic_check(void)`

我们先通过上层通用包装（`alloc_page/free_page`）各分配三页 `p0,p1,p2`，验证三者不同、引用计数为 0、物理地址在界内等基本正确性。随后**快照保存**当前 buddy 状态（每阶链表头与计数、以及全局空闲），并把所有阶链表清空、全局计数置 0 来构造一个“空系统”：在这个空系统上，`alloc_page()` 必须失败；再把刚才那三页释放回去（这一步验证释放路径在空系统上也能正确建立空闲链表与计数），此时全局空闲应为 3；再尝试分配三页，应全成功，第四页应失败；接着释放/重新分配 `p0` 验证“释放后可拿回同一页”的常见约定与链表头部行为；最终空闲变回 0。

完整自检：`buddy_check(void)`

第一步，它遍历所有阶的空闲链表，逐块检查：每个链表元素都必须 `PageProperty(p)` 为真且 `p->property==该阶`，并把各块的页数 `2^i` 累加，最终总和必须与 `buddy_nr_free_pages()` 完全一致；

随后进行如下测试：

① 记录当前全局空闲 `before`，分配 3 页 `q`，断言空闲减 3；再释放回去，空闲恢复。

② 分配 8 页 `r`，然后**错位**释放其后 5 页（`r+3` 开始），再释放前 3 页，检查系统能将这两段正确合并回整块（考验 `insert_region` 的切分对齐与伙伴合并逻辑）。

③ 构造一个“隔离小型场地”：先从全局借出 32 页 `arena`，**保存当前全局状态**，将全局链表清空/计数清零，把这 32 页用 `insert_region` 作为唯一空闲资源导入（相当于在沙箱中运行）。

在沙箱里做一系列操作：分四次各分配 4 页（`b0..b3`），再分四次各分配 4 页（`d0..d3`），释放 `b0+1` 开始的 2 页，随后尝试分配 3 页（应失败，验证“不可跨非对齐小洞拼接”的性质）；再释放 `b0` 的首页与第 4 页，使之能组合成一块 4 页整块，验证随后分配 4 页能**恰好**拿回 `b0`；陆续释放并合并所有 4 页块，再分配 8 页，拆成两段 4 页释放，再次分配 8 页应拿回同一块（验证 4→8 的向上合并完整性）。

④ 在沙箱链表的每个阶内，做“双重循环”检查：对每个块头页 `pa`，计算其伙伴索引 `idxa ^ block_size`，并在同阶链表中扫描，断言不存在一个块头 `pb` 的索引恰是这个伙伴索引（否则说明链表中存在一对可合并的块，违反不变量 ②）。

⑤ 做一次“尽可能抽干”的 1 页小块分配（到上限 64 次）：这考察在高碎片压力下仍能维持状态一致。做完所有沙箱测试后，把全局链表与计数恢复为保存的真实状态，将先借出的 32 页归还到真实系统，并检查系统全局空闲数与进入沙箱前一致。

导出接口表：`buddy_pmm_manager`

最后的 `buddy_pmm_manager` 是把本 Buddy 分配器挂接到上层物理内存管理框架（`pmm_manager` 接口）的“虚表”：`.init`、`.init_memmap`、`.alloc_pages`、`.free_pages`、`.nr_free_pages`、`.check` 分别指向前述实现。

Slub

首先是总体结构与两层分配思路。这个简化版 SLUB 由“页层（pmm）+对象层（slub）”两层组成：小到中等大小（这里为 16–2048 字节）的分配请求，会被归类到某个 size-class（16/32/64/128/256/512/1024/2048）并在该类下的 slab 中以固定大小切片分配；每个 slab 实际上就是一页（PGSIZE）被切成若干个等长对象槽位。对于超过 2048 字节（或没有合适 size-class）的大块请求，直接从页层整页（甚至多页）获取，并在用户空间前方写一个 `large_hdr_t` 头保存“魔数、页数、起始页指针”，以便未来 `kfree` 倒推页信息进行整块归还。这样的两层结构把“外部碎片”隔离到页层，把“内部碎片”控制在 size-class 的离散粒度之内，同时保持了对象层 O(1) 的快路径。

代码块

```
typedef struct slab_page {
    struct Page *page;
    void *free_head;
    unsigned int inuse;
    unsigned int capacity;
    struct slab_page *next;
} slab_page_t;

typedef struct {
    size_t object_size;
    slab_page_t *partial;
    slab_page_t *full;
} kmem_cache_t;

#define SLUB_NUM_CLASSES 8
static kmem_cache_t caches[SLUB_NUM_CLASSES];
static const size_t class_sizes[SLUB_NUM_CLASSES] = {16, 32, 64, 128, 256,
512, 1024, 2048};

typedef struct large_hdr {
    uint32_t magic;
    uint32_t pages;
    struct Page *page;
} large_hdr_t;

#define LARGE_MAGIC 0x4C52474C

#define SLUB_SP_MAX 256
static slab_page_t sp_pool[SLUB_SP_MAX];
static int sp_freelist_head = -1;
```

核心数据结构分三类。

第一类是承载一个 slab（一页）的运行时元信息 `slab_page_t`：其中 `page` 指向这页对应的 `struct Page`，`free_head` 是“对象内嵌空闲链”的表头指针，`inuse/capacity` 分别记录已用对象数与总对象数，`next` 用于把多个 slab 串成单链。值得强调的是，这个实现没有把对象之外再放独立元数据区，而是把空闲链指针直接“嵌”在对象槽位的首字（即把对象前 `sizeof(void*)` 视作下一个空闲对象的指针存储区域），这就是 SLUB 的经典做法，能让分配/释放退化成“栈顶弹出/压入”两三条指令。

第二类是每个 size-class 的缓存 `kmem_cache_t`：记录该类对象大小 `object_size`，并维护两条 slab 链 `partial`（未满）与 `full`（已满）。当一个 slab 被分配至满，需从 `partial` 迁移到 `full`；当释放使其不再满，又要从 `full` 迁回 `partial`；当释放至空（`inuse==0`），本实现选择立刻把整页归还给页层，而不是保留一个“empty”池，从而减少空转驻留。第三类是大块分配头 `large_hdr_t`：用固定魔数 `0x4C52474C`（"LRGL"），记录页数与起始页指针，释放时通过回溯用户指针前的这个头完成整页回收，避免引入外部侧表。

新建 slab 的过程由 `new_slab_page(cache)` 完成，步骤紧凑而关键：首先通过页层 `alloc_page()` 拿到一页，并用 `page2pa(pg)+PHYSICAL_MEMORY_OFFSET` 将物理地址映射到内核虚拟地址（KVA），保证随后可以直接读写。接着根据该 cache 的对象大小 `obj_size` 计算一页可容纳多少对象 `objs = PGSIZE / obj_size`；如果为 0 说明对象过大，此层不适合承载，直接释放页并失败。随后从页首开始按 `obj_size` 为步长切片，把每个对象槽位的前 `sizeof(void*)` 字节写成“前一链头地址”，用头插法把所有空闲对象串成一条单链，最终链头放入 `free_head`。最后从 `sp_pool` 取一个 `slab_page_t`，填好 `page/free_head/inuse/capacity`，并把该 slab 头插到该 cache 的 `partial` 链表作为新的“活动 slab”。整个过程没有额外 per-object 的元数据负担，只有 slab 级的常量开销（一个 `slab_page_t`），因此内存与时间都很经济。

代码块

```
static slab_page_t *new_slab_page(kmem_cache_t *cache) {
    struct Page *pg = alloc_page();
    if (pg == NULL) return NULL;
    void *mem = (void *)(page2pa(pg) + PHYSICAL_MEMORY_OFFSET);
    size_t obj_size = cache->object_size;
    size_t objs = PGSIZE / obj_size;
    if (objs == 0) {
        free_page(pg);
        return NULL;
    }
    void *head = NULL;
    for (size_t i = 0; i < objs; i++) {
        void **slot = (void **)((char *)mem + i * obj_size);
        *slot = head;
        head = slot;
    }
```

```
    slab_page_t *sp = alloc_sp();
    if (sp == NULL) {
        free_page(pg);
        return NULL;
    }
    sp->page = pg;
    sp->free_head = head;
    sp->inuse = 0;
    sp->capacity = (unsigned int)objs;
    sp->next = cache->partial;
    cache->partial = sp;
    return sp;
}
```

分配入口 `kmalloc(size)` 的第一步是用 `pick_cache(size)` 选择 size-class。这里还特意把最小对象大小提升到指针宽度（`sizeof(void*)`），保证对象内嵌指针的对齐与有效性，避免对齐错误导致的未定义行为。如果找不到合适的 class（返回 −1），说明是大块分配：把 `size` 向上按页对齐为字节数，再换算成页数 `pages_need`，调用 `alloc_pages(pages_need)`，把返回的 KVA 起点当作 `large_hdr_t` 写入 `{magic,pages,page}`，然后把“头部之后”的地址交给用户。反之命中某一类，则确保 `cache->partial` 非空（必要时 `new_slab_page` 新建），从 `partial` 头部 slab 取对象：读出 `obj = sp->free_head` 并推进 `sp->free_head = *(void**)obj`，同时 `sp->inuse++`。如果 `inuse==capacity`，说明该 slab 被用满了，就把它从 `partial` 摘下头插到 `full`。有一个小小的边界兜底：若某个 `partial` slab 的 `free_head` 意外为空（比如先前边界迁移时机造成的临时不一致），代码会把它移到 `full` 并递归再次调用 `kmalloc(size)` 选择新的活动 slab。

代码块

```
void *kmalloc(size_t size) {
    int idx = pick_cache(size);
    if (idx < 0) {
        size_t bytes = align_up(size, PGSIZE);
        size_t pages_need = bytes / PGSIZE;
        struct Page *pg = alloc_pages(pages_need);
        if (pg == NULL) return NULL;
        void *kva = (void *)(page2pa(pg) + PHYSICAL_MEMORY_OFFSET);
        if (bytes < sizeof(large_hdr_t)) {
            bytes = sizeof(large_hdr_t);
        }
        large_hdr_t *hdr = (large_hdr_t *)kva;
        hdr->magic = LARGE_MAGIC;
        hdr->pages = (uint32_t)pages_need;
        hdr->page = pg;
        return (void *)((char *)kva + sizeof(large_hdr_t));
    }
```

```
    kmem_cache_t *cache = &caches[idx];
    if (cache->partial == NULL) {
        if (new_slab_page(cache) == NULL) return NULL;
    }
    slab_page_t *sp = cache->partial;
    void *obj = sp->free_head;
    if (obj == NULL) {
        cache->partial = sp->next;
        sp->next = cache->full;
        cache->full = sp;
        return kmalloc(size);
    }
    sp->free_head = *(void **)obj;
    sp->inuse++;
    if (sp->inuse == sp->capacity) {
        cache->partial = sp->next;
        sp->next = cache->full;
        cache->full = sp;
    }
    return obj;
}
```

释放入口 `kfree(ptr)` 先尝试将指针归还给任一小块 cache 的 slab，如果都失败，再尝试走大块头部回收。小块路径依赖 `free_in_list(list_head, cache, ptr)`：它遍历 `list_head` 指向的 slab 单链（可能是 `partial`，也可能是 `full`），对每个 slab 计算它的一页 KVA 覆盖区间 `[base, base+PGSIZE)`，只要 `ptr` 落在某一页内，就把该对象“头插”回 `sp->free_head`，并在必要时进行链迁移与页回收。更具体地说：如果该 slab 来自 `full` 且释放后 `inuse<capacity`，要把它从 `full` 单链中摘下来头插回 `partial`（实现中用 `prev` 做单链删除）；如果释放后 `inuse==0`，表示 slab 完全空闲，则把它从所在链表摘除，`free_page()` 归还页层，并 `free_sp()` 归还 `slab_page_t` 元数据。这样既保证了“满↔未满”的正确迁移，又保证空 slab 不长驻内存，维持低驻留占用。若遍历完所有 cache 的 `partial` 与 `full` 都没命中，`kfree` 会把 `ptr` 回退一个 `large_hdr_t` 的大小去读头部，校验魔数与页数后用 `free_pages(hdr->page, hdr->pages)` 归还整块页，完成大块回收路径。

代码块

```
static int free_in_list(slab_page_t **list_head, kmem_cache_t *cache, void *ptr) {
    for (slab_page_t *sp = *list_head, *prev = NULL; sp != NULL; prev = sp, sp = sp->next) {
        uintptr_t base = page2pa(sp->page) + PHYSICAL_MEMORY_OFFSET;
        uintptr_t end  = base + PGSIZE;
        if ((uintptr_t)ptr >= base && (uintptr_t)ptr < end) {
            *(void **)ptr = sp->free_head;
            sp->free_head = ptr;
```

```
            if (sp->inuse > 0) sp->inuse--;
            if (list_head == &cache->full && sp->inuse < sp->capacity) {
                if (prev) prev->next = sp->next; else *list_head = sp->next;
                sp->next = cache->partial;
                cache->partial = sp;
            }
            if (sp->inuse == 0 && sp->capacity > 0) {
                if (list_head == &cache->partial) {
                    if (prev) prev->next = sp->next; else *list_head = sp->next;
                }
                free_page(sp->page);
                free_sp(sp);
            }
            return 1;
        }
    }
    return 0;
}
```

代码块

```
void kfree(void *ptr) {
    if (ptr == NULL) return;
    for (int i = 0; i < SLUB_NUM_CLASSES; i++) {
        if (free_in_list(&caches[i].partial, &caches[i], ptr)) return;
        if (free_in_list(&caches[i].full, &caches[i], ptr)) return;
    }
    large_hdr_t *hdr = (large_hdr_t *)((char *)ptr - sizeof(large_hdr_t));
    if (hdr->magic == LARGE_MAGIC && hdr->page != NULL && hdr->pages > 0) {
        free_pages(hdr->page, hdr->pages);
    }
}
```

关于不变量与复杂度，这个实现保持了几个重要关系：对任一 slab，始终有 `0 ≤ inuse ≤ capacity`；空闲链长度与 `capacity - inuse` 一致；`partial` 链上所有 slab 都满足“未满”，`full` 链上所有 slab 都是“已满”。常见分配与释放操作都是对单链头的弹/压，时间复杂度 O(1)；只有在 `new_slab_page` 与“空 slab 释放”两个时机会触达页层，因此摊还开销低。当作教学或嵌入式内核分配器的雏形，这种复杂度足以支撑大量小对象的高频分配回收。

`slub_init` 一次性把 `sp_pool` 串成空闲索引栈，并为每个 cache 设置对象大小与初始空链表；`slub_check` 做了三次典型的小块分配（24/200/1024），覆盖不同 size-class，并在释放后检验 `partial/full` 的迁移是否正确。由于 1024 仍在 2048 之内，这三次都走小块路径，能覆盖 slab

的创建、填充、迁移与回收的常见流程。如果你想测试大块路径，只需请求一个超过 2048 的尺寸，比如 `kmalloc(3000)`，就会触发 `large_hdr_t` 头的写入与按页回收。

代码块

```
void slub_init(void) {
    sp_freelist_head = -1;
    for (int i = SLUB_SP_MAX - 1; i >= 0; i--) {
        sp_pool[i].next = (slab_page_t *)(uintptr_t)sp_freelist_head;
        sp_freelist_head = i;
    }
    for (int i = 0; i < SLUB_NUM_CLASSES; i++) {
        caches[i].object_size = class_sizes[i];
        caches[i].partial = NULL;
        caches[i].full = NULL;
    }
}

```

硬件的可用物理内存范围的获取方法（思考题）

1. 手动探测物理内存：

一种操作上比较简单的方式是逐块地访问内存，记录下可用内存的起止范围。

例如，逐个向内存块中写入测试数据，并读取；同时设计相应的异常捕获机制。当系统尝试访问超出物理内存范围的地址时，引发异常（如段错误等），OS需要能够捕捉到异常，以此来确定内存的边界。

2. 利用外设间接探测：

通过与外围设备交互来间接推测物理内存范围：可以通过配置DMA控制器，指定内存地址和数据长度，让DMA执行内存操作。若数据传输失败，则该内存地址可能无效或不存在。