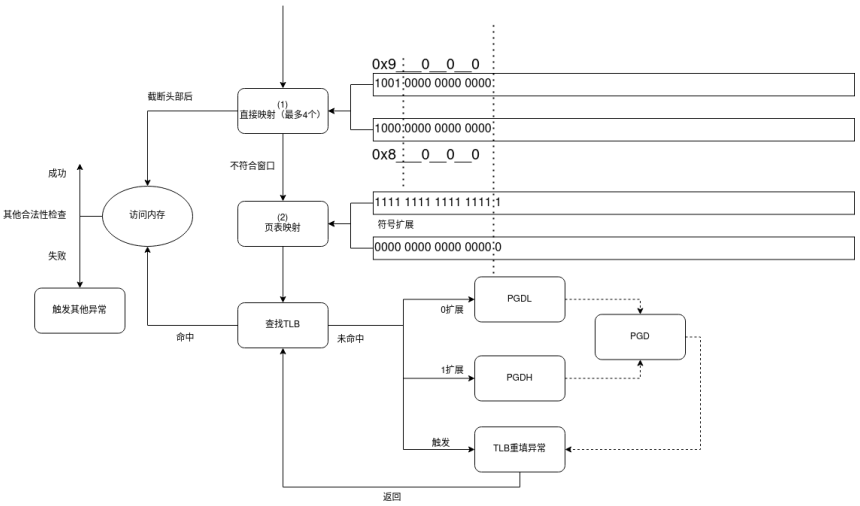
目录

1 NPUcore+ LA 的内存模块适配

1.1 内存地址映射布局

LoongArch 的虚拟映射模式内存机理如下图:



首先，检查是否符合直接映射窗口（通过 DMW0 ~ 3 四个 CSR 进行映射），如果其高 4 位相同则认为是符合，则将其他位数截断，作为物理地址访问

其次，如果不是，则检查是否符号扩展，是则尝试查找 TLB，否则出发异常，在 miss 后，如果是 1 扩展则 PGD 为 PGDH，0 扩展为 PGDL，

然后触发 TLB 重填异常，ertn 返回后重新进行 TLB 查找。如果此时没有对应的 TLB，则触发页无效异常。

此外, 对 DMW 的权限是: 0 号和 1 号窗口是 RWX, 2 号和 3 号窗口是 RW(不可执行). 每个窗口可以单独设置自己的缓存一致性类型

由于 LoongArch 的特殊特性，NPUcore+ LA 内存布局采取了和 NPUcore RISC-V 下不同的方案：

1. 对用户态，使用 0 扩展地址作为虚拟空间，
2. 对内核态，用 0 扩展地址空间访问物理内存，然后对页表映射使用 1 扩展的地址，

这样有下列好处：

1. 可以用直接继承原版 NPUcore 的部分内存布局，无需为了利用 LA 的直接映射窗口就大量修改代码,在物理地址上按位或大量的 0x9000000000000000，减少出错空间

2. 可以利用直接映射减小恒等映射的开销

3. 在上下文无需存储页表地址，可以直接切换地址空间。

1.1.1 跳转

从 u-boot 获得控制权时,引导程序提供了几个: 已经映射好的段: 0x9000 段 (一致可缓存), 和 0x8000 段 (无缓存直接访问内存), 前者使用 DMW1, 后者 DMW0(可能随 u-boot 版本不同而不同); 我们的代码加载从 0 物理地址开始 (见 start.pdf 文件的叙述)

这时候的 PC 是 0x9000 段的 0 地址, 但基于之前的分析, 我们

由于地址空间中 0 段地址是不被映射的，因此需要某些方法先设置 DMW 映射 0 段，再让 PC 跳转到该地址（NPUcore 内核态的 PC 是在物理地址上的）。为此, 我们的启动代码 entry.asm 需要进行如下设计:

1. 在跳转到主事件循环 rust_main 时, 需要保证后续

具体代码如下:

```
_start:
    pcaddi      $t0,    0x0
    srli.d      $t0,    $t0,    0x30
    slli.d      $t0,    $t0,    0x30 ;位移删去物理地址
    addi.d      $t0,    $t0,    0x11 ;计算当前的窗口CSR值
    csrwr       $t0,    0x181
;上面代码保证窗口DMW0写入切换后不会被覆盖,所以先将DMW1设为当前段
    sub.d       $t0,    $t0,    $t0
    addi.d      $t0,    $t0,    0x11 ;计算0段DMW的值
    csrwr       $t0,    0x180  ; 设置0段DMW的值
    sub.d       $t0,    $t0,    $t0;$t0=0
```

```
jirl        $t0,    $t0,    0x28 ;跳转到下一条指令的绝对地址
# The barrier
sub.d       $t0,    $t0,    $t0 ;$t0=0
csrwr       $t0,    0x181 ;写入DMW1, 清零DMW1
sub.d       $t0,    $t0,    $t0 ;清零t0
la.global $sp, boot_stack_top ;加载boot_stack_top
bl          rust_main ;跳转到rust_main
```

这段代码的注意点如下:

1. 注意这里没有使用 $t0=$zero 是因为在 QEMU 虚拟机下, 某些时候似乎 $zero 寄存器会被赋值为 0(GDB 提示), 因此使用手工计算的 t0=0

2. 先设置映射窗口, 而不是计算完 0 段的 DMW 控制寄存器数值后直接覆盖到 DMW0 的原因如果当前 PC 的地址处在 DMW0 而非 DMW1 的区段内 (这个由引导程序 u-boot 决定), 覆盖 DMW0 后 PC 的地址就会成为非法地址

   而跳转必然发生在映射了新的区段号后

   因此要先将当前的区段保存到 DMW1, 然后再进行对 DMW0 的配置, 从而确保至少有一个 DMW 是 PC 当前的地址.

1.2 不对齐读写问题

1.2.1 问题由来与不同开发平台的支持状态

我们知道, 部分的数据结构依赖于, 例如:

另一方面, 当前开源的 LLVM 的 LoongArch 后端不支持生成严格对齐的代码, 因此也导致依赖 LLVM 后端进行代码生成的 Rustc 无法编译出可以在开发板上正常运行的代码, 这也就要求我们对 NPUcore 的不对齐异常手动处理.

虽然 QEMU 支持不对齐读写, 但是开发板是无法支持不对齐读写的. 如果要在 QEMU 上支持不对齐读写的检测和不对齐读写的报错, 需要开启环境变量 DEBUG_UNALIGN=1, 否则会忽略所有的不对齐读写异常. 具体的启动命令如下:

```
DEBUG_UNALIGN=1 SERIAL=2 DEBUG_GMAC_PHYAD=0 \
DEBUG_MYNAND=cs=0,id=0x2cda \
DEBUG_MYSPIFLASH=gd25q128 qemu-system-loongarch64 ...
```

具体见 util/qemu-2k500/gz/runqemu2k500.

如果将来 LoongArch 的 LLVM 完成了严格对齐读写的修复, 则应当可以用 target-feature 开启编译器的选项:

```
[target.loongarch64-unknown-linux-gnu]
rustflags = ["-Ctarget-feature=-unaligned-access",
          "-Clink-arg=-Tsrc/linker.ld", "-Clink-arg=-nostdlib", "-Clink-arg=-static"]
```

这个问题在 C 语言的 GCC 下是不存在的, 因此, 理论上在 Rust 编写的操作系统上, 当前会由于不对齐读写存在性能瓶颈. 为了解决这个问题, 我们需要在内核态手动模拟不对齐读写指令的执行.

1.2.2 内核态的解决

首先是内核态的不对齐异常. 这种异常实际上只出现在栈上, 因为静态数据和堆上的数据结构是对齐的, 而 NPUcore-LA 的内核堆则是从栈上分配的, 只有在 ELF 加载阶段有部分只读临时的内核页表映射. 因此, 只需要保存内容后, 对进行逐字节解读取即可.

具体来说, 为了节省内容恢复的空间, 我们直接对 kernel trap 的设计为:

1. 内容保存不需要单独建立一个位置, 直接将寄存器上下文保存在栈上, 因为内核的不对齐读写异常是发生在执行时, 所以直接视为一个单独的函数调用即可

2. 指令解码使用

内容保存的代码如下 (来自 trap.S):

```
__kern_trap:
    # Keep the original $sp in SAVE
    csrwr $sp, CSR_SAVE
    csrrd $sp, CSR_SAVE
    # Now move the $sp lower to push the registers
    addi.d $sp, $sp, -256
    # Align the $sp
    srli.d  $sp, $sp, 3
    slli.d  $sp, $sp, 3
    # now sp->*GeneralRegisters in kern space, CSR_SAVE->(the previous $sp)

    SAVE_GP 1 # Save $ra
    SAVE_GP 2 # Save $tp

    # skip r3(sp)
    .set n, 4
    .rept 28
        SAVE_GP %n
        .set n, n+1
    .endr
    .set n, 0
    csrrd $t0, CSR_ERA
    st.d $t0, $sp, 0

    move $a0, $sp
    csrrd $sp, CSR_SAVE
    st.d $sp, $a0, 3*8
    move $sp, $a0

    bl trap_from_kernel

    ld.d  $ra, $sp, 0
    csrwr $ra, CSR_ERA
    LOAD_GP 1
    LOAD_GP 2

    # skip r3(sp)
    .set n, 4
    .rept 28
        LOAD_GP %n
        .set n, n+1
    .endr
    .set n, 0

    csrrd $sp, CSR_SAVE
    ertn
```

而读写的关键代码如下 (见 os/src/arch/la64/trap/mod.rs:440)

```
if op.is_store() {
    let mut rd = gr[ins.get_rd_num()];
    for i in 0..sz {
        unsafe { ((addr + i) as *mut u8).write_unaligned(rd as u8) };
        rd >>= 8;
    }
} else {
```

```
    let mut rd = 0;
    for i in (0..sz).rev() {
        rd <<= 8;
        let read_byte =
            (unsafe { ((addr + i) as *mut u8).read_unaligned() } as usize);
        rd |= read_byte;
        //debug!("{:#x}, {:#x}", rd, read_byte);
    }
    if !op.is_unsigned_ld() {
        match sz {
            2 => rd = (rd as u16) as i16 as isize as usize,
            4 => rd = (rd as u32) as i32 as isize as usize,
            8 => rd = rd,
            _ => unreachable!(),
        }
    }
    gr[ins.get_rd_num()] = rd;
}
gr.pc += 4;
```

然后按照 Rust 的类型扩展为目标位数即可

注意这里的 rev 是必须的, 因为 LoongArch 是小端机器, 所以高字节的先读才能被或到低位

理论上, 还存在更快的办法, 就是判断是否跨整 2^{n} 字节, 如果不跨过, 则用汇编选择对应字节的内容, 否则就读入两寄存器 (128bits) 再取各自需要的部分组合成一个寄存器的值. 但由于其实现较为复杂, 所以我们没有采用.

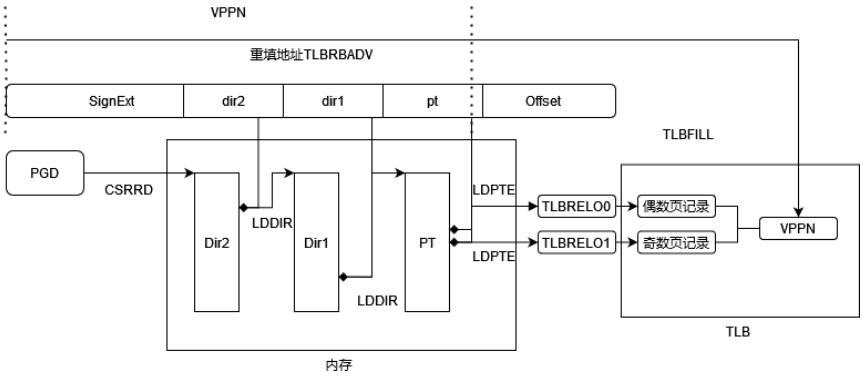
1.2.3 用户态的解决

用户态的解决方案会更复杂一些, 这主要是因为用户态的不对齐读写可能会跨越多个物理页, 所以直接按照偏移量读入到目标寄存器在的时候会产

生部分的问题. 为此, 我们可以使用 NPUcore 自带的 copy_from_user 函数进行逐字节的拷贝, 从而屏蔽从用户态读取数据的跨页问题.

1.3 TLB refill 与页表结构

1.3.1 TLB 重填异常



一旦在虚拟地址模式中被判定为页表映射的地址, 发生 TLB miss, 就会触发 TLB 重填异常。LoongArch 目前是手动重填 TLB 的。其重填的一般流程如下:

1. 硬件提供页表地址 PGD
   (a) LoongArch 有两个页表 PGDL 和 PGDH，分别对应符号扩展的全 1 段和 0 段。
   (b) 如果触发重填的地址来自 0 段, 则重填的页表 PGD 此时等于 PGDL, 否则 PGD 此时等于 PGDH。这样一来, 页表实际上分为了高地址和低地址页表, 可以用作不同的功能。
2. 硬件触发 TLB 异常，跳转到 TLBRENTRY CSR 所指向的地址
3. 操作系统会响应异常，通过读取 PGD 状态控制寄存器, 获取最高一级页表目录

4. 根据虚拟地址计算出对应的物理页框号。然后在主存中查找该物理页框的对应页表项，然后返回该页表项的物理地址. 为了加速页表重填, LoongArch 提供了下列特性:

   (a) LoongArch 的 TLB 以双页形式组织，目的是减少重填次数。具体来说，LoongArch 的 TLB 的索引单位是 VPPN（Virtual Page Pair Number), 为虚拟页号去掉最第一位。每个 VPPN 对应 VPN 为 VPPN×2+VPN[12] 的一对奇数页和偶数页两页的物理地址。考虑到 TLB 重填的局部性，这样最多可以减少一半的 TLB 重填。

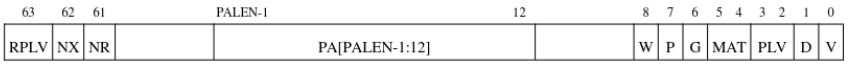
   (b) LoongArch 提供 LDDIR 和 LDPTE 两种指令进行页表遍历，通过 TLBRBADV 寄存器提供重填的虚拟地址，并以此为基础获得各级页表的索引号。LDDIR 是用于给定非末级页表起始地址，求取本级页表项（或者说下一级页表起始地址），其格式为 (其中 imm 为页表级数。以 rj 为页表起始地址，rd 为目的寄存器。)：
   LDDIR rd, rj, imm,

5. 返回页表项后，处理器将该对页表项用 TLBFILL 写入 TLB 中，以便下次使用该虚拟地址时，能够直接从 TLB 中获取物理地址信息，而不需要再次触发 TLB 重填。在 TLB 更新后，处理器会重新执行之前的指令或内存访问操作，这次操作可以直接从 TLB 中获取到物理地址

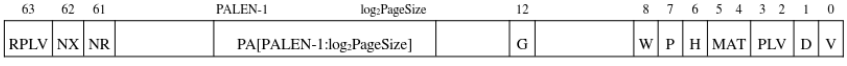
1.3.2 LoongArch 的页表结构

官方手册对 LoongArch 的项目并没有清晰的叙述, 其中存在部分不明确的地方. 具体来说, 除了大页之外, LoongArch 并不规定页目录的页表项格式, 所以其实际上是未定义的.

基本页页表项格式：



大页页表项格式：



(图片来自龙芯手册)

而上文提到的 LDDIR 和 LDPTE 并不检查目录项合法性, 且对非法的页表项仍会继续寻址, 因此, 对目录项需要有手工的检测或者非法目录项的处理方法.

1.3.3 直接模拟

最简单的方法是直接模拟其他 RISC 架构的页表处理方式, 直接用汇编代码对各层页表项进行处理. 首先, 页表是一棵前缀树, 其部分没有被映射的节点是空的, 因此如果直接使用上述的 LDPTE 和 LDDIR, 由于其只是单纯的将异常地址的对应段取出作相加, 因此对空的地址会计算出错误的地址, 而不是和其他 RISC 一样触发异常. 因此, 非最底层页表的叶结点 (空表项) 必须要手工判断, 并对其错误的表项, 填写无读写权限的 TLB 表项

```
rfill:
    csrwr   $t0, 0x8b ;将寄存器$t0暂存到TLB重填例外数据保存（0x8b）
    csrrd   $t0, 0x1b ;读取此时的页表根节点PGD（0x1b）内容
    lddir   $t0, $t0, 3 ;读取第3层页表的页表项
    andi    $t0, $t0, 1;取valid位
    beqz    $t0, 1f;检查valid位， 如果invalid则跳转

    csrrd   $t0, 0x1b;读取此时的页表根节点PGD（0x1b）内容
    lddir   $t0, $t0, 3;读取第3层页表的页表项
    addi.d  $t0, $t0, -1;去除valid位
    lddir   $t0, $t0, 1;读取第2层页表的页表项
    andi    $t0, $t0, 1;取valid位
    beqz    $t0, 1f;检查valid位， 如果invalid则跳转
    csrrd   $t0, 0x1b;读取此时的页表根节点PGD（0x1b）内容
    lddir   $t0, $t0, 3;读取第3层页表的页表项
    addi.d  $t0, $t0, -1;去除valid位
    lddir   $t0, $t0, 1;读取第2层页表的页表项
    addi.d  $t0, $t0, -1;去除valid位

    ldpte   $t0, 0;读取偶页表项
    ldpte   $t0, 1;读取奇页表项
2:
    tlbfill;填写缺失的TLB
    csrrd   $t0, 0x8b;恢复$t0的内容
    ertn ;从异常返回
1:
    csrrd   $t0, 0x8e ;读取TLBREHI
    ori     $t0, $t0, 0xC ;计算填入的页表大小为4K
    csrwr   $t0, 0x8e ;写回

    add.d   $t0, $zero, $zero ;计算空项目
    rotri.d $t0, $t0, 61;循环位移高3位到低3位， 方便或立即数
    ori     $t0, $t0, 3;规定该页无效且不可读， 不可写
    rotri.d $t0, $t0, 3;循环位移，恢复该页表项的排序
    csrwr   $t0, 0x8c; 填入偶数页
    csrrd   $t0, 0x8c; 重新读取偶数页项
    csrwr   $t0, 0x8d;填入奇数页
    b       2b;跳回无效页填写
```

上述代码实现了以下功能:

1. 逐层读取页表项, 如果没到最后一层, 检查读取到的页表项的合法性,
   (a) 合法则继续读取下一级页表项
   (b) 非法则准备填入 0 页表项, 表示该页非法
2. 填入 0 页表项或者读取最后一层页表项结束后, 将页表大小填写为

4KiB, 然后向 TLB 填入 0 地址, 表示该地址不合法。注意, 这段代码有 2 个需要注意的细节:

(a) 由于该段代码使用的 csrwr 指令在 LoongArch 下的语义是交换 CSR 和寄存器的内容, 而非简单地将寄存器内容写入, 因此在向两个相同结构的 CSR 填入某个相同数值的时候, 我们需要重新读取之前的目的寄存器, 然后重新写入才能正确处理结果。

(b) 之所以需要对 TLB Refill exception Entry HIgh-order bits (TLBREHI)(0x8e) 的状态控制寄存器填入 0, 是因为该地址内包括页长度相关的域, 但该地址原本是由 ldpte 填写, 为了防止手动填写 TLB 项导致未定义行为 (填入错误的 TLB 或虚拟机报错), 我们需要先对该页填写 0 地址