



中国科学技术大学
University of Science and Technology of China

University of Science and Technology of China

loongarch xv6

An Manual for the loongarch

loongarch xv6

2023 年 5 月 23 日

修改的文件

修改的文件

bio.c

```
1
2 #include "riscv.h" -> #include "loongarch.h"
3
4 virtio_disk_rw(b, 0); -> ramdiskrw(b, 0);
5
6 virtio_disk_rw(b, 1); -> ramdiskrw(b, 1);
```

暂未搞懂的文件

exec.c

loongreg.h 和 memlayout.h 中有重复的宏定义。

proc.c 中进程栈的分配结构变化

内存管理

0.1 虚实地址翻译

0.1.1 loongarch 的虚实地址翻译

我们需要改的第一个地方就是虚实地址翻译，可以利用 loongarch 架构特性来对于 risc-v 版本的 xv6 进行优化，提高性能。

原理介绍

建议先阅读《loongarch 指令集手册》第五章，了解 loongarch 的虚实地址翻译机制。

阅读完成后，我们讲解一下其中的重点。

在 LoongArch 架构中，MMU 支持两种虚拟地址到物理地址的转换模式：**直接地址转换模式**和**映射地址转换模式**

我们重点关注**映射地址转换模式**

映射地址转换模式，有两种类型的地址转换模式：**直接映射地址转换模式（直接映射模式）**和**页表映射地址转换模式（页表映射模式）**。在转换地址时，优先使用直接映射模式。只有在直接映射模式无法转换地址时，才使用页表映射模式进行转换。

在 LA64 中，在使用**页表映射模式**时，虚拟地址空间合法性的规则如下：合法虚拟地址的 [63:PALEN] 位必须与 [PALEN-1] 位相同，否则将触发地址错误异常（ADE）。然而，在直接映射模式下，不需要进行此地址合法性检查。

我们看到，在间接映射模式下，页表映射模式的合法规则非常的严格，这就意味着很多的地址无法使用页表映射模式进行转换。但是，我们可以利用这个特性，来实现一些特殊的功能。比如在 riscv 版本中，内核和用户空间都是利用页表来进行地址转换的，而在 loongarch 中，我们可以**利用直接映射模式来实现内**

核空间的地址转换，而利用页表映射模式来实现用户空间的地址转换。

这样做有很多好处：

- 在内核使用直接映射窗口时不需要页表，从而节省内存。
- 不会产生 TLB 重填例外，从而加快虚实地址翻译速度。
- 简化设计，在内核和用户空间切换时不需要做页表的转换，提高性能。

代码修改

前面提到，利用直接映射模式来实现内核空间的地址转换，而利用页表映射模式来实现用户空间的地址转换，那么我们需要一段无法用页表映射模式进行地址转换的地址空间，用这一段空间来实现内核空间的地址转换。如果 PALEN 等于 48，且 DMWO 被设置为 $0x9000000000000011^1$ ，则虚拟地址空间 $0x9000000000000000-0x9000FFFFFFFFFFFFFF$ 将直接映射到物理地址空间 $0x0-0xFFFFFFFFFFFFFF$

所以我们在 memlayout.h 里面定义

Code 1: memlayout.h

```
1 #ifdef __ASSEMBLY__
2 #define _CONST64_(x)      x
3 #else
4 #define _CONST64_(x)      x ## L
5 #endif
6
7 #define DMW_PABITS    48
8
9 #define CSR_DMW1_PLV0    _CONST64_(1 << 0)
10 #define CSR_DMW1_MAT     _CONST64_(1 << 4)
11 #define CSR_DMW1_VSEG    _CONST64_(0x9000)
12 #define CSR_DMW1_BASE    (CSR_DMW1_VSEG << DMW_PABITS)
13 #define CSR_DMW1_INIT    (CSR_DMW1_BASE | CSR_DMW1_MAT |
    CSR_DMW1_PLV0)
```

这段代码定义了一些宏，用来设置 DWM1 寄存器的值。

- 将物理地址 $0x0-0xFFFFFFFFFFFFFF$ 映射到虚拟地址 $0x9000000000000000-0x9000FFFFFFFFFFFFFF$

¹详见龙芯手册 7.5.18

- 将 PALEN 设置为 48

我们现在已经设置了地址转换模式，我们的目的是把它作为内核空间的地址转换模式，所以我们需要把内核的代码和数据放在这段地址空间中。

我们把这一步目标放在 kernel.ld 文件上

Code 2: kernel.ld

```
1 OUTPUT_ARCH( "loongarch" )
2 ENTRY(_entry)
3
4 SECTIONS
5 {
6     /*
7      * ensure that entry.S / _entry is at 0x9000000000000000,
8      * where qemu's -kernel jumps.
9      */
10    . = 0x9000000000000000;
11    .text : {
12        ...
13    }
14    .rodata : {
15        ...
16    }
17    .data : {
18        ...
19    }
20    .bss : {
21        ...
22    }
23    PROVIDE(end = .);
24 }
```

我们让内核从**虚拟地址** 0x9000000000000000 开始，这样就可以利用直接映射模式来进行地址转换了，我们初步的目的就达成了。

UEFI bios 装载内核时，实际跳转到的地址将是 0x0，这也是内核的**物理**起始地址。

而我们现在需要设置 DMW1 寄存器，来实现用户空间的地址转换，我们在 entry.S 中设置 DMW1 寄存器

Code 3: entry.S

```
1      li.d      $t0, CSR_DMW1_INIT
2      csrwr     $t0, 0x181
```

这样我们就完成了直接映射窗口设置。

我们设置直接映射窗口前，没有地址映射，所以 PC 只能直接使用物理地址，也就是 0x0 0xFFFFFFFFFFFFFFF，而我们设置了直接映射窗口后，PC 就应该使用虚拟地址，我们应该按照我们设置的规则来设置 PC，也就是让 PC 使用 0x9000000000000000 0x9000FFFFFFFFFFFFFFF 这段地址空间。

我们在 entry.S 中设置 PC

Code 4: entry.S

```
1      # 计算ertn后一条指令的虚拟地址
2      li.d      $t0, CSR_DMW1_BASE
3      pcaddi    $t1, 0x5           # -----+
4      add.d     $t0, $t0, $t1      #         |
5                                   #         |
6      # 将虚拟地址写入tlbrera      #         |
7      # 并设置tlbrera.IsTLBR      #         |
8      ori $t0, $t0, 0x01          #         |
9      csrwr     $t0, 0x8a         #         |
10     ertn      #         |
11              #         |
12     la.abs    $sp, stack0+4096   # <-----+
```

这里首先算出 extn 后一条指令的虚拟地址，也就是 PC+CSR_DMW1_BASE+5，然后将这个虚拟地址写入 tlbrera 寄存器，也就是在 tlb 重填例外后的返回地址²，这样 tlb 重填例外返回后，PC 就会跳转到这个虚拟地址，我们也就达到了设置 PC 的目的。

值得一提的是，我们之所以要利用例外来设置 PC，是因为在某些处理器中，没有提供直接设置 PC 的指令，只能通过 branch 等指令来设置 PC，但是，也有一些架构支持直接写入 PC，但是 loongarch 不支持，所以我们通过例外来设置 PC。

那到目前为止，我们已经做到了我们之前设想的把内核的代码和数据放在直接映射窗口的目的，但是，我们现在还没有把所有的内核代码中使用了物理地址的地方都改成了虚拟地址，所以我们还需要做一些工作。

²参考龙芯手册表 7.1

下面就需要把所有使用了物理地址的地方都改成虚拟地址。

举例，在 `kalloc` 函数中，我们使用了物理地址，我们需要把它改成虚拟地址。

Code 5: `kalloc.c`

```
1 void kfree(void *pa)
2 {
3     struct run *r;
4     if((uint64)pa <= MAX_USED_PHYSADDR)
5         pa = P2V(pa);
6
7     if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)
8         pa >= P2V(PHYSTOP))
9         panic("kfree");
10
11     // Fill with junk to catch dangling refs.
12     memset(pa, 1, PGSIZE);
13
14     r = (struct run *)pa;
15
16     acquire(&kmem.lock);
17     r->next = kmem.freelist;
18     kmem.freelist = r;
19     release(&kmem.lock);
20 }
```

需要修改的地方就不一一列出。

我们可以把 `vm.c` 中那些和内核页表相关的删除了，因为我们不采用内核页表了。

这一部分完成。

0.1.2 loongarch 的页表

原理介绍

在做这一部分前，需要先阅读龙芯手册 5.4 节内容。

xv6-riscv 版采用三级页表，而 loongarch 采用了多级页表，因此需要对 xv6-riscv 版的页表进行修改。

xv6-loongarch 版的页表结构如下图所示。

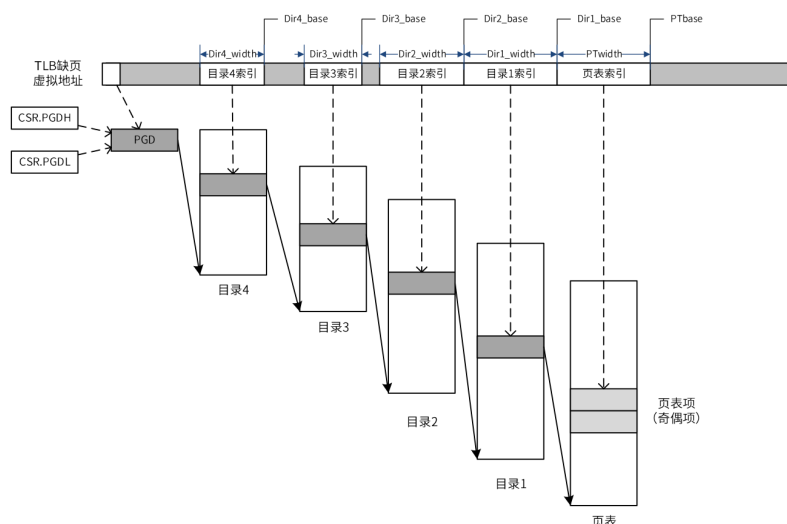


图 1: 多级页表结构

我们定义了虚拟地址共 48 位，在 loongarch 中使用页表时，如果 $VA[47] = 0$ ，使用 CSR.PGDL 作为目录基址。如果 $VA[47] = 1$ ，使用 CSR.PGDH 作为目录基址。剩下的 47 位用于遍历页表。

xv6-longarch 只使用了 PGDL 相当于只有 47 位虚拟地址，这 47 位虚拟地址全留给用户空间使用，内核使用映射窗口。

```

1  A 64-bit virtual address is split into seven fields:
2  48..63 -- must be zero.
3  47      -- must be zero, (only PGDL used).
4  39..46 -- 8 bits of level-3 index.
5  30..38 -- 9 bits of level-2 index.
6  21..29 -- 9 bits of level-1 index.
7  12..20 -- 9 bits of level-0 index.
8  0..11  -- 12 bits of byte offset within the page.

```

这样就把这 47 为虚拟地址分割成了 $8 + 9 + 9 + 9 + 12$ 这种格式，使用的是四级页表，页大小为 4KB。

0.1.3 代码修改

修改 walk 函数

walk 函数的作用是根据虚拟地址找到对应的页表项，如果 alloc 为 1，那么如果页表项不存在，就分配一个页表项。在 xv6-ricsv 中，使用三级页表，而在 loongarch 中，使用四级页表，代码修改如下：

Code 6: walk()

```
1 pte_t * walk(pagetable_t pagetable, uint64 va, int alloc)
2 {
3     ...
4     for(int level = 3; level > 0; level--) {
5         pte_t *pte = &pagetable[PX(level, va)];
6         pte = (pte_t *)P2V((char *)pte);
7         ...
8     }
9     return &pagetable[PX(0, va)];
10 }
```

还有代码中一些利用到了页表标志的地方，需要修改。比如：

Code 7: freewalk()

```
1 void freewalk(pagetable_t pagetable)
2 {
3     // there are 2^9 = 512 PTEs in a page table.
4     pagetable = (pagetable_t)P2V((char*)pagetable);
5     for(int i = 0; i < 512; i++){
6         pte_t pte = pagetable[i];
7         if( pte && !(pte & 0xff)){
8             ...
9         } else if (!pte)
10             ...
11     }
12     kfree((void*)pagetable);
13 }
```

0.1.4 用户空间内存布局

在 xv6-loongarch 中，第一个和 xv6-riscv 中不同的是，**选择将栈放在虚拟内存的顶端**。第二个点是，trampoline 的作用是在用户态和内核态之间切换页表，而之前提到过，loongarch 的一个优势就是在内核使用直接映射窗口进行映射，也就是不需要切换页表，所以在用户空间中不需要分出一块空间来给 trampoline。

xv6-riscv	xv6-loongarch
trampoline	trapframe
trapframe	stack
heap	stack guard page
.....	heap
stack	
stack guard page	
data and bss	data and bss
text	text

图 2: 进程在用户空间的内存布局

根据上面的图，我们修改 `proc.c` 中的 `proc_pagetable` 函数，也就是负责分配进程空间的函数，代码如下：

Code 8: `proc.c`

```

1 pagetable_t proc_pagetable(struct proc *p)
2 {
3     pagetable_t pagetable;
4     uint64 x;
5
6     // An empty page table.
7     pagetable = uvmcreate();
8     if(pagetable == 0)
9         return 0;
10
11
12     // map the trapframe in MAXV-PAGESIZE, for trampoline.S.
13     if(mappages(pagetable, TRAPFRAME, PGSIZE,
14                 (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
15         uvmfree(pagetable, 0);

```

```

16     return 0;
17 }
18
19 if((x = (uint64)kalloc()) == 0) {
20     uvmfree(pagetable, 0);
21     return 0;
22 }
23
24 // map the userstack in USTACK-PGSIZE
25 if(mappages(pagetable, USTACK-PGSIZE, PGSIZE,
26             x, PTE_R | PTE_W) < 0){
27     uvmunmap(pagetable, TRAPFRAME, 1, 0);
28     uvmfree(pagetable, 0);
29     return 0;
30 }
31 return pagetable;
32 }

```

以及修改负责释放用户空间的代码 `proc_freepagetable`:

需要强调, 一开始按照释放 `trampframe` 的做法释放 `ustack`, 也就是,

`uvmunmap(pagetable, USTACK-PGSIZE, 1, 0);`

`ustack` 内存并没有释放, 因为在 `freeproc()` 中会释放 `trampframe` 占用的内存, 所以传递给 `uvmunmap()` 的最后一个参数为 0 (只清 0 页表项, 不释放内存), 但是 `ustack` 并没有在 `freeproc()` 中被释放

所以应把释放 `ustack` 的 `uvmunmap()` 函数最后一个参数设置为 1(将页表项清 0 的同时释放页表项对应的物理页面).

Code 9: `proc.c`

```

1 void proc_freepagetable(pagetable_t pagetable, uint64 sz)
2 {
3     uvmunmap(pagetable, USTACK-PGSIZE, 1, 1);
4     uvmunmap(pagetable, TRAPFRAME, 1, 0);
5     uvmfree(pagetable, sz);
6 }

```

由于初始的 `xv6` 代码数据和栈相连 (中间隔了一个保护页), `p->sz` 包含栈, 所以 `fork()` 时会通过 `vmcopy()` 复制栈中的内容, 新的实现中 `p->sz` 不包含栈需要在 `vmcopy()` 中添加代码把栈中的内容也复制过来。 `uvmcopy()` 添加代码如下:

Code 10: vm.c

```
1 uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
2 {
3     ...
4     uint64 oldstack_pa, newstack_pa;
5     if((pte = walk(old, USTACK-PGSIZE, 0)) == 0)
6         panic("uvmcopy: pte should exist");
7     pte = (pte_t *)P2V((char *)pte);
8     oldstack_pa = PTE2PA(*pte);
9     if((pte = walk(new, USTACK-PGSIZE, 0)) == 0)
10        panic("uvmcopy: pte should exist");
11    pte = (pte_t *)P2V((char *)pte);
12    newstack_pa = PTE2PA(*pte);
13    memmove((char*)newstack_pa, (char*)oldstack_pa, PGSIZE);
14    ...
15 }
```

对于各个进程的内核栈，直接在 procinit() 中分配并且不再设置 guardpage。
(TODO)

Code 11: proc.c

```
1 // initialize the proc table at boot time.
2 void
3 procinit(void)
4 {
5     struct proc *p;
6
7     initlock(&pid_lock, "nextpid");
8     initlock(&wait_lock, "wait_lock");
9     for(p = proc; p < &proc[NPROC]; p++) {
10         initlock(&p->lock, "proc");
11         // p->kstack = KSTACK((int) (p - proc));
12         p->kstack = (uint64)kalloc();
13     }
14 }
```
