

内存

这里只介绍 xv6-riscv 和我队所做的 xv6-loongarch 有关内存方面的区别，以及 loongarch 给系统程序员提供的接口。

目录

1. 直接映射窗口 CSR.DMW0---CSR.DMW3 (0x180---1x183):.....	2
1.1 什么是直接映射窗口?	2
1.2 直接映射窗口命中规则.....	2
1.3 xv6-longarch 中的实现.....	2
1.4 直接映射窗口的优点.....	4
2. 页表.....	5
2.1 CSR.PWCL 和 CSR.PWCH.....	5
3. xv6-riscv 与 xv6-loongarch 在内存布局上的区别.....	7
3.1 User.....	7
3.2 Kernel.....	10

1. 直接映射窗口 CSR.DMW0---CSR.DMW3 (0x180---1x183):

1.1 什么是直接映射窗口？

直接映射窗口是一种将虚拟内存**线性映射**到物理内存的手段, 用一个简单的数学函数描述即:

$$y = x - b$$

这里 y 表示物理地址, x 表示虚拟地址, b 为一常量表示为虚拟内存所命中的映射窗口的 Base 值, Base 值由映射窗口的 VSEG 域决定.

$$\text{Base} = \text{VSEG} \ll 60$$

1.2 直接映射窗口命中规则

LA64 下 $\text{VA}[63:60] = \text{VSEG}$. 并且特权等级符合。

若把 CSR.DMW1.VSEG 设置为 9, 则会把 0x9000000000000000-0x9000ffffffff 这段虚拟地址映射到 0x0-0xffffffff 这段物理地址.

1.3 xv6-longarch 中的实现

1. 修改 kernel/kernel.ld 将内核的链接地址为 0x9000000000000000:

```
1 OUTPUT_ARCH("loongarch")
2 ENTRY(_entry)
3
4 SECTIONS
5 {
6     /*
7      * ensure that entry.S / _entry is at 0x9000000000000000,
8      */
9     . = 0x9000000000000000;
10
11     .text : {
12         *(.text .text.*)
13         . = ALIGN(0x1000);
14         trampoline = .;
15         *(trampsec)
16         . = ALIGN(0x1000);
17         ASSERT(. - _trampoline == 0x1000, "error: trampoline larger than one page");
18         PROVIDE(etext = .);
19     }
20 }
```

2. 由于 UEFI bios 装载内核时, 会把从内核 elf 文件获取的入口点地址 (可以用 `readelf -h` 或者 `-l kernel` 看到) 抹去高 32 位使用。比如 kernel 链接的地址是 0x9000000000000000, 实际 bios 跳转的地址将是 0x0, 代码装载的位置也是物理

内存 0x0。BIOS 这么做是因为它逻辑上相当于用物理地址去访问内存，高的虚拟地址空间没有映射不能直接用。

所以内核启动入口代码需要做两件事：

- i. 设置一个直接地址映射窗口把内核用到的 64 地址抹去高位映射到物理内存。目前 linux 内核是设置 0x8000xxxx-xxxxxxx 和 0x9000xxxx-xxxxxxx 地址抹去最高的 8 和 9 为其物理地址，前者用于 uncached 访问(即不通过高速缓存去 load/store)，后者用于 cache 访问。**xv6 内核只用到了后者。**
- ii. 做个代码自跳转，使得后续代码执行的 PC 和链接用的虚拟地址匹配。

BIOS 刚跳转到内核时，用的地址是抹去了高 32 位的地址（相当于物理地址），步骤 1 使得链接时的高地址可以访问到同样的物理内存，这里则换回到原始的虚拟地址。

实现如下：

```
22      # 使用映射窗口映射整个物理内存
23      # 通过模拟一个tlb例外来实现
24      li.d    $t0, CSR_DMW1_INIT
25      csrwr   $t0, 0x181
26
27      # 设置例外前模式
28      li.w    $t0, 0x00          # PPLV=0, PPIE=0, PPWE=0
29      csrwr   $t0, 0x8b
30
31      # 计算ertn后一条指令的虚拟地址
32      li.d    $t0, CSR_DMW1_BASE
33      pcaddi   $t1, 0x5          # -----|
34      add.d    $t0, $t0, $t1     #         |
35                                   #         |
36      # 将虚拟地址写入tlbrera   #         |
37      # 并设置tlbrera.IsTLBR    #         |
38      ori     $t0, $t0, 0x01     #         |
39      csrwr   $t0, 0x8a         #         |
40      ertn                                   #         |
41                                   #         |
42      la.abs   $sp, stack0+4096  # <-----|
43      csrrd    $t0, 0x20        # CPUID
```

3. 更改 xv6-riscv 中所有使用物理地址的 C 代码，转换为使用虚拟地址(pa+DMW1_base).

- i. 将 UART0 控制寄存器基址由 0x1FE001E0 改为 0x900000001FE001E0
- ii. 虽然内核不需要使用页表，但是用户空间需要使用页表。由于页目录项和页表项必须要使用物理地址，所以内核遍历页表时(比如 walk()函数)特别需要注意虚实地址的转换。比如 walk():

```

25 pte_t *
26 walk(pagetable_t pagetable, uint64 va, int alloc)
27 {
28     if(va >= MAXVA)
29         panic("walk");
30
31     for(int level = 3; level > 0; level--) {
32         pte_t *pte = &pagetable[PX(level, va)];
33         pte = (pte_t *)P2V((char *)pte);
34         //if(*pte & PTE_V) {
35         if(*pte) {
36             pagetable = (pagetable_t)PTE2PA(*pte);
37         } else {
38             if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
39                 return 0;
40             memset(pagetable, 0, PGSIZE);
41             //pte = PA2PTE(pagetable) | PTE_V | PTE_MAT_CC;
42             pagetable = (pagetable_t)V2P((char *)pagetable);
43             *pte = PA2PTE(pagetable);
44         }
45     }
46     return &pagetable[PX(0, va)];

```

32 行代码获取的指针为页目录项的物理地址，不能直接用*pte 获取页目录项的内容，**需要将 pte 转化为对应的虚拟地址使用(第 33 行代码)**。另外 mappages(),walkaddr(),uvmunmap(),freewalk(),uvmcopy(),uvmclear()函数内都需要类似转换。

iii. xv6-riscv 内核中 string.c 中的函数接收的一般都是物理地址，所以需要转换为虚拟地址使用。

iv. xv6-riscv 中 kalloc()函数返回的是页面的物理地址，xv6-riscv 中很多代码会直接使用返回的物理地址读写内存，如果修改所有调用 kalloc()的地方改动就会太大，不如直接把 kalloc()的返回地址改为虚拟地址。但是 xv6-riscv 中创建页表时会把 kalloc()的返回地址传递给 mappages()，而且 mappages()接受的物理地址参数全部为 kalloc()的返回值，所以直接修改 mappages()让其接受虚拟地址：

```

77 int
78 mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
79 {
80     uint64 a, last;
81     pte_t *pte;
82     pa = V2P(pa);
83
84     if(size == 0)
85         panic("mappages: size");
86

```

由于 kalloc()返回虚拟地址，所以 kfree()必须可以接受虚拟地址。

1.4 直接映射窗口的优点

- i. 使用直接映射窗口时不需要页表，从而节省内存。
- ii. 不会产生 TLB 重填例外，从而加快虚实地址翻译速度。
- iii. 简化内核设计。比如从用户空间 trap 后进入内核，此时使用的是用户空

间的页表所以必须让内核的页表和用户空间的页表的同一虚拟地址映射到 trap 入口处。而且需要做页表的切换。如果使用映射窗口就不需要在用户页表映射到 trap 入口处，而且也不需要做页表的转换(可以看作 trap 进入内核的同时自动切换到内核页表)。

2. 页表

loongarch 下 PGSIZE 可由系统程序员自由设置。

2.1 CSR.PWCL 和 CSR.PWCH

这两个寄存器一起定义了系统的页表结构，一开始在这两个寄存器上的 BASE 域上迷惑了很久，所以举例详细说明以下这两个寄存器的用法。下面是 CSR.PWCL 和 CSR.PWCH 的定义：

表 7-29 页表遍历控制低半部分寄存器定义

位	名字	读写	描述
4:0	PTbase	RW	末级页表的起始地址。
9:5	PTwidth	RW	末级页表的索引位数。
14:10	Dir1_base	RW	最低一级目录的起始地址。
19:15	Dir1_width	RW	最低一级目录的索引位数。0 表示没有这一级。
24:20	Dir2_base	RW	次低一级目录的起始地址。
29:25	Dir2_width	RW	次低一级目录的索引位数。0 表示没有这一级。
31:30	PTEWidth	RW	内存中每个页表项的位宽。 0 表示 64 比特，1 表示 128 比特，2 表示 192 比特，3 表示 256 比特。

表 7-30 页表遍历控制高半部分寄存器定义

位	名字	读写	描述
5:0	Dir3_base	RW	次高一级目录的起始地址。
11:6	Dir3_width	RW	次高一级目录的索引位数。0 表示没有这一级。
17:12	Dir4_base	RW	最高一级目录的起始地址。
23:18	Dir4_width	RW	最高一级目录的索引位数。0 表示没有这一级。
31:24	0	RO	保留域。读返回 0，且软件不允许改变其值。

如果使用页表，虚拟地址共 48 位。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。**xv6-longarch** 将这 47 位分成了五部分如下：

```
17 // A 64-bit virtual address is split into seven fields:
18 //   48..63 -- must be zero.
19 //   47      -- must be zero, (only PGDL used).
20 //   39..46 -- 8 bits of level-3 index.
21 //   30..38 -- 9 bits of level-2 index.
22 //   21..29 -- 9 bits of level-1 index.
23 //   12..20 -- 9 bits of level-0 index.
24 //    0..11 -- 12 bits of byte offset within the page.
```

此时 CSR.PWCL.PTbase = 12, CSR.PWCL.PTwidth = 9
CSR.PWCL.Dir1_base = 21, CSR.PWCL.Dir1_width = 9
CSR.PWCL.Dir2_base = 30, CSR.PWCL.Dir2_width = 9
CSR.PWCL.PTEWidth = 0, 页表项和页目录项均为 64 位
CSR.PWCH.Dir3_base = 39, CSR.PWCH.Dir3_width = 8
CSR.PWCH 其余位为 0.

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。也可以使用 16KB 的页面，此时 47 位虚拟地址应划分为 11 + 11 + 11 + 14，使用三级页表，这时候 CSR.PWCH 应设置位 0，CSR.PWCL 设置如下：

CSR.PWCL.PTbase = 14, CSR.PWCL.PTwidth = 11
CSR.PWCL.Dir1_base = 25, CSR.PWCL.Dir1_width = 11
CSR.PWCL.Dir2_base = 36, CSR.PWCL.Dir2_width = 11

即 VA[0:13]为页内偏移，VA[14:24]为页表索引，VA[25:35]为目录 1 索引，VA[36:47]为目录 2 索引。**xv6-longarch** 使用的是前者的 4 级索引，页大小 4KB。

3. xv6-riscv 与 xv6-loongarch 在内存布局上的区别

3.1 User

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

xv6-riscv 中 stack guard page 是拥有实际的物理内存的，为了减少一页物理内存的消耗，而且不改变 `sys_brk` 的行为，所以 xv6-loongarch 选择将栈尽量放在虚拟内存的顶端。用户空间栈的分配从 `exec()` 转移到 `allocproc->proc_pagtable()`。

问题：

i 由于初始的 xv6 代码数据和栈无缝链接，`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：

```
268 }
269 uint64 oldstack_pa, newstack_pa;
270 if((pte = walk(old, USTACK-PGSIZE, 0)) == 0)
271     panic("uvmcopy: pte should exist");
272 pte = (pte_t *)P2V((char *)pte);
273 oldstack_pa = PTE2PA(*pte);
274 if((pte = walk(new, USTACK-PGSIZE, 0)) == 0)
275     panic("uvmcopy: pte should exist");
276 pte = (pte_t *)P2V((char *)pte);
277 newstack_pa = PTE2PA(*pte);
278 memmove((char*)newstack_pa, (char*)oldstack_pa, PGSIZE);
```

ii 然后 `exec_test()` 报错，原因是 `sys_exec()->fetchaddr()` 会检查地址范围不能超过 `p->sz`：


```

1 int
2 fetchaddr(uint64 addr, uint64 *ip)
3 {
4     struct proc *p = myproc();
5     if(addr >= p->sz || addr+sizeof(uint64) > p->sz)
6         return -1;
7     if(copyin(p->pagetable, (char *)ip, addr, sizeof(*ip)) != 0)
8         return -1;
9     return 0;
10 }
11

```

由于把栈设置在虚拟地址次高页处，而且 `exectest()` 把参数设置在了栈中：

```

672 void
673 exectest(char *s)
674 {
675     int fd, xstatus, pid;
676     char *echoargv[] = { "echo", "OK", 0 };
677     char buf[3];
678
679     unlink("echo-ok");
680     pid = fork();
681     if(pid < 0) {
682         printf("%s: fork failed\n", s);
683         exit(1);
684     }
685     if(pid == 0) {
686         close(1);
687         fd = open("echo-ok", O_CREATE|O_WRONLY);
688         if(fd < 0) {
689             printf("%s: create failed\n", s);
690             exit(1);
691         }
692         if(fd != 1) {
693             printf("%s: wrong fd\n", s);
694             exit(1);
695         }
696         if(exec("echo", echoargv) < 0){
697             printf("%s: exec failed\n", s);
698             exit(1);
699         }
700     }
701     if(pid > 0) {
702         wait(&xstatus);
703         if(xstatus < 0) {
704             printf("%s: wait failed\n", s);
705             exit(1);
706         }
707         if(xstatus > 0) {
708             printf("%s: child died\n", s);
709             exit(1);
710         }
711     }
712 }
713

```

`p->sz` 不包含栈，故 `addr > p->sz`，故返回-1。所以应把 `fetchaddr()` 改为：

```

11 int
12 fetchaddr(uint64 addr, uint64 *ip)
13 {
14     struct proc *p = myproc();
15     if((addr >= p->sz && addr < USTACK-PGSIZE) ||
16        (addr+sizeof(uint64) > p->sz && addr < p->sz))
17         return -1;
18     if(copyin(p->pagetable, (char *)ip, addr, sizeof(*ip)) != 0)
19         return -1;
20     return 0;
21 }
22

```

iii 改完后发现会出现释放进程时无法全部释放进程拥有的内存。


```

216 void
217 proc_freepagetable(pagetable_t pagetable, uint64 sz)
218 {
219     uvmunmap(pagetable, USTACK-PGSIZE, 1, 0);
220     uvmunmap(pagetable, TRAPFRAME, 1, 0);
221     uvmfree(pagetable, sz);
222 }

```

一开始按照释放 `trampfram` 的做法释放 `ustack`，`ustack` 内存并没有释放，每次执行一个进程都少两页（一个在 `exec()` 中调用 `proc_freepagetable()` 没有被释放，一个在退出时调用 `proc_freepagetable()` 没有被释放）。

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）。

```

215 // physical memory it refers to.
216 void
217 proc_freepagetable(pagetable_t pagetable, uint64 sz)
218 {
219     uvmunmap(pagetable, USTACK-PGSIZE, 1, 1);
220     uvmunmap(pagetable, TRAPFRAME, 1, 0);
221     uvmfree(pagetable, sz);
222 }
223

```

3.2 Kernel

VA

xv6-riscv	xv6-loongarch
trampoline	mem(DMW1_base+0x90000000)
NULL
process0 kstack	
process0 kstack guard page	
.....	
process63 kstack	
process63 kstack guard page	
kernel(0x80000000)	kernel(DMW1_base)

PA-riscv

```
6 // 00001000 -- boot ROM, provided by qemu
7 // 02000000 -- CLINT
8 // 0C000000 -- PLIC
9 // 10000000 -- uart0
10 // 10001000 -- virtio disk
11 // 80000000 -- boot ROM jumps here in machine mode
12 //                -kernel loads the kernel here
13 // unused RAM after 80000000.
14 #define KERNBASE 0x80000000L
15 #define PHYSTOP (KERNBASE + 128*1024*1024)
16
```

PA-loongarch

```
71 const MemMapEntry virt_memmap[] = {
72     [VIRT_LOWMEM] = { 0x00000000, 0x10000000 },
73     [VIRT_PM] = { 0x10080000, 0x100 },
74     [VIRT_FW_CFG] = { 0x10080100, 0x100 },
75     [VIRT_RTC] = { 0x10081000, 0x1000 },
76     [VIRT_PCIE_PIO] = { 0x18000000, 0x80000 },
77     [VIRT_PCIE_ECAM] = { 0x1a000000, 0x2000000 },
78     [VIRT_BIOS_ROM] = { 0x1fc00000, 0x200000 },
79     [VIRT_UART] = { 0x1fe001e0, 0x8 },
80     [VIRT_LIOINTC] = { 0x3ff01400, 0x64 },
81     [VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
82     [VIRT_HIGHMEM] = { 0x90000000, 0x0 }, /* Variable */
83 };
84
85 static const MemMapEntry loader_memmap[] = {
86     [LOADER_KERNEL] = { 0x00000000, 0x4000000 },
87     [LOADER_INITRD] = { 0x04000000, 0x0 }, /* Variable */
88     [LOADER_CMDLINE] = { 0x0ff00000, 0x100000 },
89 };
```

由于内核使用了映射窗口，所以内核不需要页表映射到 **trampoline**,对于各个进程的内核栈，直接在 **procinit()**中分配并且不再设置 **guardpage**。

```
34 void
35 procinit(void)
36 {
37     struct proc *p;
38
39     initlock(&pid_lock, "nextpid");
40     initlock(&wait_lock, "wait_lock");
41     initlock(&nproc_lock, "nproc_lock");
42     for(p = proc; p < &proc[NPROC]; p++) {
43         initlock(&p->lock, "proc");
44         // p->kstack = KSTACK((int) (p - proc));
45         p->kstack = (uint64)kalloc();
46     }
47 }
```