



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 6 月 7 日

Summary

1	内存管理	1
1.1	虚实地址翻译	1
1.1.1	loongarch 的虚实地址翻译	1
1.1.2	loongarch 的页表	6
1.1.3	代码修改	7
1.1.4	用户空间内存布局	8
2	中断	12
2.1	基本概念	12
2.1.1	Riscv 中断	12
2.1.2	Loongarch 中断	13
2.1.3	移植前的基本定义	15
2.2	Loongarch 例外移植	16
2.2.1	普通例外移植	17
2.2.2	TLB 重填例外	19
2.2.3	机器错误例外	20
2.2.4	中断移植	20
3	文件系统	22
3.1	基本概念	22
3.1.1	文件系统结构	22
3.1.2	Buffer cache	23

3.1.3	logging	26
3.1.4	Inode	27
3.1.5	directory	28
3.1.6	Path names	28
3.1.7	File descriptor	29
4	调度	31
4.1	基本概念	31
4.1.1	Context switching	31
4.1.2	Scheduling	32
4.1.3	代码修改	32

Chapter 1

内存管理

1.1 虚实地址翻译

1.1.1 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 0x9000000000000011¹，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 1.1: 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)
```

¹详见龙芯手册 7.5.18

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 0x0-0xFFFFFFFF 映射到虚拟地址 0x9000000000000000-0x9000FFFFFFFFFFFFFF
- 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 1.2: 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 1.3: 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 0xFFFFFFFFFFFFFFFF，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 0x9000FFFFFFFFFFFFFFFF 这段地址空间。

我们在 entry.S 中设置 PC

Code 1.4: 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

²参考龙芯手册表 7.1

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。

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 1.5: 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。

1.1.2 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。

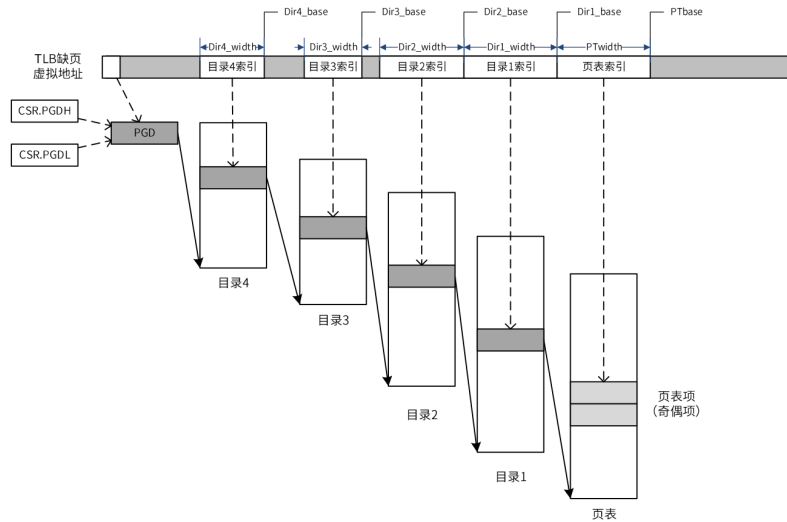


图 1.1: 多级页表结构

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。

1.1.3 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-riscv 中，使用三级页表，而在 loongarch 中，使用四级页表，代码修改如下：

Code 1.6: 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 1.7: 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 }
```

1.1.4 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 1.2: 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 1.8: 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 `trampfram` 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 1.9: `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 1.10: `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 1.11: `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 }
```

Chapter 2

中断

2.1 基本概念

2.1.1 Riscv 中断

在 xv6-riscv 中中断分为三类：系统调用/自陷（system call）、故障/异常（exception）和硬件中断（interrupt）

中断的原则：对源程序透明，意味着源程序的通用寄存器、PC、状态（包括运行模式等等）必须有保存和恢复。

Riscv 的中断例外寄存器

- stvec：中断发生后 PC 跳转的地址。
- sepc：中断发生时的 PC，在 sret 后 PC 跳转向 sepc。
- scause：描述中断原因。
- sscratch：在中断处理最开始使用的寄存器，在 xv6 中断机制里保存着本进程 trapframe 的地址。
- sstatus：状态寄存器，SIE 字段是是否允许硬件中断，SPP 字段是中断前的模式，在 sret 时将模式改为 SPP。

Riscv 中断流程

用户态中断：ecall（指令，硬件完成）->uservec（汇编）->usertrap（C）->(syscall, devintr, ...)->usertrapret（C）->userret（汇编）->sret（指令，硬件完成）。

- ecall: 保存断点到 epc、保存中断前模式 (spp)、记录中断原因 (scause)、关中断 (sie), 跳转到 stvec。
- uservec: 保存通用寄存器、用户栈到 TRAPFRAME, 恢复内核栈、内核线程指针、内核页表, 跳转到 usertrap。
- usertrap: 保存 epc 到 trapframe, 恢复内核中断处理函数-> 开中断, 具体处理中断-> 跳转到 usertrapret。
- usertrapret: 关中断, 恢复用户中断处理函数, 保存内核栈、内核线程指针、内核页表, 修改 SPIE 为开中断、修改 SEPC 为断点。
- userret: 恢复用户页表, 恢复通用寄存器、用户栈, 保存 trapframe 地址。
- sret: 恢复 pc 为 epc, sie 为 spie (中断), 模式为 spp (这里为用户模式)。

2.1.2 Loongarch 中断

龙芯架构下的中断采用线中断的形式, 每个处理器核内部可记录 13 个线中断, 分别是: 1 个核间中断, 1 个定时器中断, 1 个性能检测计数溢出中断, 8 个硬中断, 2 个软中断

核间中断来源于核外的中断控制器, 其被处理器核采样记录在 CSR.ESAT.IS[12] 位

定时器中断来自于核内的恒定频率定时器, 当恒定频率定时器倒计时至全 0 时, 该中断被置起。定时器中断被处理器核采样记录在 CSR.ESAT.IS[11] 位

硬中断来自处理器核外部, 其直接来源通常是核外的中断控制器, 8 个硬中断 HWI[7:0] 被处理器采样记录在 CSR.ESAT.IS[9:2]

软中断来源于处理器核内部, 软件通过 CSR 指令对 CSR.ESAT.IS[1:0] 写 1 则置起软中断, 写 0 则清除软中断

例外的主要寄存器 (详见龙芯架构参考手册)

龙芯架构下定义了一系列控制状态寄存器 (Control and Status Register, 简称 CSR), 用于控制指令的执行行为, 下面的 CSR.%%%.#### 的形式用来指称名称缩写为%%%. 的控制状态寄存器中的名字为 #### 的域

例外前模式信息 (PRMD)

当出现例外时, 如果例外类型不是 TLB 重填或机器错误例外, 硬件会将此时处理器核的特权等级、全局中断使能和监视点使能位保存至例外前模式信息

寄存器中

表 7-3 例外前模式信息寄存器定义

位	名字	读写	描述
1:0	PPLV	RW	当触发例外时，如果例外类型不是 TLB 重填例外和机器错误例外，硬件会将 CSR.CRMD 中 PLV 域的旧值记录在这个域。 当所处理的例外既不是 TLB 重填例外（CSR.TLBRERA.IsTLBR=0）也不是机器错误例外（CSR.ERRCTL.IsMERR=0）时，执行 ERTN 指令从例外处理程序返回时，硬件会将这个域的值恢复到 CSR.CRMD 的 PLV 域。
2	PIE	RW	当触发例外时，如果例外类型不是 TLB 重填例外和机器错误例外，硬件会将 CSR.CRMD 中 IE 域的旧值记录在这个域。 当所处理的例外既不是 TLB 重填例外（CSR.TLBRERA.IsTLBR=0）也不是机器错误例外（CSR.ERRCTL.IsMERR=0）时，执行 ERTN 指令从例外处理程序返回时，硬件会将这个域的值恢复到 CSR.CRMD 的 IE 域。
3	PWE	RW	当触发例外时，如果例外类型不是 TLB 重填例外和机器错误例外，硬件会将 CSR.CRMD 中 WE 域的旧值记录在这个域。 当所处理的例外既不是 TLB 重填例外（CSR.TLBRERA.IsTLBR=0）也不是机器错误例外（CSR.ERRCTL.IsMERR=0）时，执行 ERTN 指令从例外处理程序返回时，硬件会将这个域的值恢复到 CSR.CRMD 的 WE 域。
31:4	0	R0	保留域。读返回 0，且软件不允许改变其值。

例外状态 (ESTAT)

该寄存器记录例外的状态信息，包括所触发例外的一二级编码，以及各中断的状态

表 7-7 例外状态寄存器定义

位	名字	读写	描述
1:0	IS[1:0]	RW	两个软件中断的状态位。比特 0 和 1 分别对应 SWI0 和 SWI1。 软件中断的设置也是通过这两位完成，软件写 1 置中断写 0 清中断。
12:2	IS[12:2]	R	中断状态位。其值为 1 表示对应的中断置起。1 个核间中断（IPI），1 个定时器中断（TI），1 个性能计数器溢出中断（PMI），8 个硬中断（HWI0~HWI7） 在线中断模式下，硬件仅是逐拍采样各个中断源并将其状态记录与此。此时对于所有中断须为电平中断的要求，是由中断源负责保证，并不在此处维护。
15:13	0	R0	保留域。读返回 0，且软件不允许改变其值。
21:16	Ecode	R	例外类型一级编码。触发例外时： 如果是 TLB 重填例外或机器错误例外，该域保持不变； 否则，硬件会根据例外类型将表 7-8 中 Ecode 栏定义的数值写入该域。
30:22	EsubCode	R	例外类型二级编码。触发例外时： 如果是 TLB 重填例外或机器错误例外，该域保持不变； 否则，硬件会根据例外类型将表 7-8 中 EsubCode 栏定义的数值写入该域。
31	0	R0	保留域。读返回 0，且软件不允许改变其值。

例外程序返回地址 (ERA)

该寄存器记录普通例外处理完毕之后的返回地址。当触发例外时，如果例外类型既不是 TLB 重填例外也不是机器错误例外，则触发例外的指令的 PC 将被记录在该寄存器中

表 7-9 例外程序计数器寄存器定义

位	名字	读写	描述
GRLEN-1:0	PC	RW	触发例外时： 如果是 TLB 重填例外或机器错误例外，该域保持不变； 否则，硬件会将触发例外的指令的 PC 记录到这里。对于 LA64 架构，在这种情况下，如果触发例外的特权等级处于 32 位地址模式，那么记录的 PC 值的高 32 位强制置为 0。

2.1.3 移植前的基本定义

首先添加 Loongarch 相关特权寄存器的定义，其中 SAVE0,1 用于入口地址的存储（数据保存）

Code 2.1: loongarchregs.h

```
1 #define LOONGARCH_CSR_CRMD      0x0      /* Current mode  
   info */  
2 #define LOONGARCH_CSR_PRMD      0x1      /* Prev-exception  
   mode info */  
3 #define LOONGARCH_CSR_CPUID      0x20     /* CPU core id */  
4  
5 #define LOONGARCH_CSR_SAVE0      0x30     /* Kscratch  
   registers */  
6 #define LOONGARCH_CSR_SAVE1      0x31     /* Kscratch  
   registers */
```

其次修改 trampframe 中的相关定义，例如寄存器的存储位置等

Code 2.2: trapframe

```
1 /* 0 */ uint64 ra;  
2 /* 8 */ uint64 tp;  
3 /* 16 */ uint64 sp;  
4 /* 24 */ uint64 a0;  
5 /* 32 */ uint64 a1;  
6 /* 40 */ uint64 a2;
```

```

7  /* 48 */ uint64 a3;
8  /* 56 */ uint64 a4;
9  /* 64 */ uint64 a5;
10 /* 72 */ uint64 a6;
11 /* 80 */ uint64 a7;
12 /* 88 */ uint64 t0;
13 /* 96 */ uint64 t1;
14 /* 104 */ uint64 t2;
15 /* 112 */ uint64 t3;
16 /* 120 */ uint64 t4;
17 /* 128 */ uint64 t5;
18 /* 136 */ uint64 t6;
19 /* 144 */ uint64 t7;
20 /* 152 */ uint64 t8;
21 /* 160 */ uint64 r21;
22 /* 168 */ uint64 fp;
23 /* 176 */ uint64 s0;
24 /* 184 */ uint64 s1;
25 /* 192 */ uint64 s2;
26 /* 200 */ uint64 s3;
27 /* 208 */ uint64 s4;
28 /* 216 */ uint64 s5;
29 /* 224 */ uint64 s6;
30 /* 232 */ uint64 s7;
31 /* 240 */ uint64 s8;
32 /* 248 */ uint64 kernel_sp;      // top of process's kernel
    stack
33 /* 256 */ uint64 era;           // saved user program counter
34 /* 264 */ uint64 kernel_hartid; // saved kernel tp
35 /* 272 */ uint64 kernel_pgdl;   // saved kernel pagetable

```

2.2 Loongarch 例外移植

例外种类分为 **TLB 重填例外**，**机器错误例外**，**普通例外**（普通例外又分为用户态例外和内核态意外），他们的区别在于例外的入口不同

TLB 重填例外的入口来自于 CSR.TLBREENTRY

机器错误例外的入口来自于 CSR.MEMREENTRY

除上述两种例外之外的例外称为普通例外，其入口地址采用”入口页号 | 页内偏移”的计算方式，其中”|”是按位或运算

所有普通例外入口的入口页号相同，均来自于 CSR.EENTRY

2.2.1 普通例外移植

普通例外硬件处理通用过程，触发普通例外时，处理器硬件进行如下主要操作

- 将 CSR.CRMD 的 PLV、IE 分别存到 CSR.PRMD 的 PPLV、PIE 中，然后将 CSR.CRMD 的 PLV 设置为 0,IE 设置为 0;
- 将触发例外的指令的 PC 记录到 CSR.ERA 中
- 跳转到例外入口取指
当软件执行 ERTN 指令从普通例外执行返回时，处理器硬件会完成如下操作
- 将 CSR.PRMD 中的 PPLV、PIE 值恢复到 CSR.CRMD 的 PLV、IE 中
- 跳转到 CSR.ERA 所记录的地址处取指

普通例外分为内核态例外和用户态例外,区分两者的主要方式是查看 CSR.CRMD.PLV 来查看优先级。在 xv6riscv 中，使用 kernelec.s 与 tramponline.S 分别处理内核例外与用户态例外，我们移植时进行了入口的统一，即通过判断 PLV 即优先级来判断是用户态例外还是内核态例外，跳转到对应的处理接口，分别使用 kernelvec.S 与 uservec.S 处理相应的例外。

添加处理例外的统一接口，即 handle_excep(), 放在 exception.S 中，下面的代码将 t0 寄存器放在 0x30 位置中，在后续的例外中，需要交换回来

```
1 .section eentry
2 .globl handle_excep
3 .align 0x4
4 handle_excep:
5     csrwr    $t0, LOONGARCH_CSR_SAVE0
6     csrrd    $t0, LOONGARCH_CSR_PRMD
7     andi     $t0, $t0, 0x3
8     beqz     $t0, kernelvec
9     b        uservec
```

内核态例外移植

在 kernelvec.S 和 uservec.S 中，将对应的汇编指令转换为龙芯架构的汇编指令即可

<pre>1 sd ra , 40(a0) 2 sd sp , 48(a0) 3 sd gp , 56(a0) 4 sd tp , 64(a0) 5 sd t0 , 72(a0) 6 sd t1 , 80(a0) 7 sd t2 , 88(a0) 8 sd s0 , 96(a0) 9 sd s1 , 104(a0) 10 sd a1 , 120(a0) 11 sd a2 , 128(a0) 12 sd a3 , 136(a0) 13 sd a4 , 144(a0) 14 sd a5 , 152(a0) 15 ...</pre>	<pre>1 st.d \$ra , \$a0 , 40 2 st.d \$tp , \$a0 , 48 3 st.d \$sp , \$a0 , 56 4 st.d \$a1 , \$a0 , 72 5 st.d \$a2 , \$a0 , 80 6 st.d \$a3 , \$a0 , 88 7 st.d \$a4 , \$a0 , 96 8 st.d \$a5 , \$a0 , 104 9 st.d \$a6 , \$a0 , 112 10 st.d \$a7 , \$a0 , 120 11 st.d \$t0 , \$a0 , 128 12 st.d \$t1 , \$a0 , 136 13 st.d \$t2 , \$a0 , 144 14 st.d \$t3 , \$a0 , 152 15 ...</pre>
图 2.1: Riscv	图 2.2: Loongarch

图 2.3: 汇编指令的修改

而在 trap.c 中，由于两种中断使用的相同的入口，所以不再需要 trapinithart 函数来表明内核例外处理的地址，直接使用 handle_excep 地址来处理即可

用户态例外的移植

由于不需要做页表切换，所以 trampoline 不再被需要，将 trampoline.S 重命名为 uservec.S 用于做用户态的例外

用户态中断流程为 uservec，然后是 usertrap；返回时是 usertrapret，然后是 userret.

用户态的修改和内核态例外相似，将汇编架构从 riscv 换为 Loongarch 架构即可

2.2.2 TLB 重填例外

发生 TLB 重填例外时，处理器进行以下操作：

- 将 CSR.CRMD 的 PLV、IE 分别存到 CSR.PRMD 的 PPLV、PIE 中，然后将 CSR.CRMD 的 PLV 设置为 0,IE 设置为 0，DA 设置为 1，PG 设置为 0；
- 将触发例外的指令的 PC[GRLEN-1:2] 记录到 CSR.TLBRERA 的 ERA 域中，将 CSR.IsTLBR 设置为 1
- 将触发例外的访存虚地址记录到 CSR.TLBRBADV 中,将虚地址的 [PALEN-1:13] 位记录到 CSR.TLBREHI 的 VPPN 域中
- 跳转到 CSR.TLBRENTTRY 所配置的例外入口处取指
当软件执行 ERTN 指令从 TLB 重填例外执行返回时，处理器硬件会完成如下操作
- 将 CSR.TLBPRMD 中的 PPLV、PIE 值恢复到 CSR.CRMD 的 PLV、IE 中
- 将 CSR.CRMD 的 DA 设置为 0,PG 设置为 1
- 将 CSR.TLBRERA 的 IsTLBR 设置为 0
- 跳转到 CSR.TLBRERA 所记录的地址处取指

在内核运行之前，需要将 tlb 初始化，增加 tlbinit 函数，用于初始化

```
1 void tlbinit(void)
2 {
3     asm volatile("invtlb 0x0,$zero,$zero");
4     w_csr_stlbps(0xcU);
5     w_csr_asid(0x0U);
6     w_csr_tlbreh(0xcU);
7 }
```

增加一个用于处理 TLB 重填的汇编代码，放在 tlbrefill.S 中，增加接口地址 handle_tlbr 处理上述流程

```
1 .section tlbrentry
2 .globl handle_tlbr
3 .align 0x4
```

```

4 handle_tlbr:
5     csrwr    $t0, LOONGARCH_CSR_TLBRSAVE
6     csrrd    $t0, LOONGARCH_CSR_PGD
7     lddir    $t0, $t0, 3
8     addi.d   $t0, $t0, -1
9     lddir    $t0, $t0, 2
10    addi.d   $t0, $t0, -1
11    lddir    $t0, $t0, 1
12    addi.d   $t0, $t0, -1
13    ldpte    $t0, 0
14    ldpte    $t0, 1
15    tlbfill
16    csrrd    $t0, LOONGARCH_CSR_TLBRSAVE
17    ertn

```

2.2.3 机器错误例外

我们将机器错误例外当成不可逆的意外，即当触发机器错误例外时，直接 panic，不再做详细处理

添加 merror.S 文件来，增加接口地址 handle_merror 处理

```

1 .section merrentry
2 .globl machine_trap
3 .globl handle_merr
4 .align 0x4
5 handle_merr:
6     bl machine_trap
7     ertn

```

2.2.4 中断移植

当发生中断时，各中断源发来中断信号被处理器采样至 CSR.ESAT.IS 域中，这些软件与信息配置在 CSR.ECFG.LIE 域中的局部中断使能信息按位与，得到一个中断向量，来判断是哪个部分的中断

在 xv6 中，中断的处理在 devintr 函数中，由于只运行在 loongarch-qemu 上，所以中断来源只来源于串口

则当中断源判断是内部中断时，直接设置为串口读写中断就可完成基本目的

Chapter 3

文件系统

3.1 基本概念

3.1.1 文件系统结构



图 3.1: 文件系统结构

文件系统的实现分为七层，如图所示。disk 层在磁盘上读写块。Buffer cache 缓存磁盘块，并同步访问它们，确保一个块只能同时被内核中的一个进程访问。日志层允许上层通过事务更新多个磁盘块，并确保在崩溃时，磁盘块是原子更新的（即全部更新或不更新）。inode 层将一个文件都表示为一个 inode，每个文件包含一个唯一的 i-number 和一些存放文件数据的块。目录层将实现了一种特殊的 inode，被称为目录，其包含一个目录项序列，每个目录项由文件名称和 i-number 组成。路径名层提供了层次化的路径名，可以用递归查找解析他们。文

件描述符层用文件系统接口抽象了许多 Unix 资源（如管道、设备、文件等），使程序员的生产力得到大大的提高。

3.1.2 Buffer cache

基本介绍

buffer 缓存有两项工作:(1) 同步访问磁盘块，以确保磁盘块在内存中只有一个 buffer 缓存，并且一次只有一个内核线程能使用该 buffer 缓存；(2) 缓存使用较多的块，这样它们就不需要从慢速磁盘中重新读取。代码见 bio.c。

buffer 缓存的主要接口包括 bread 和 bwrite，bread 返回一个在内存中可以读取和修改的块副本 buf，bwrite 将修改后的 buffer 写到磁盘上相应的块。内核线程在使用完一个 buffer 后，必须通过调用 brelse 释放它。buffer 缓存为每个 buffer 的都设有 sleep-lock，以确保每次只有一个线程使用 buffer（从而使用相应的磁盘块）；bread 返回的 buffer 会被锁定，而 brelse 释放锁。

buffer 缓存有固定数量的 buffer 来存放磁盘块，这意味着如果文件系统需要一个尚未被缓存的块，buffer 缓存必须回收一个当前存放其他块的 buffer。buffer 缓存为新块寻找最近使用最少的 buffer（lru 机制）。因为最近使用最少的 buffer 是最不可能被再次使用的 buffer。

实现

buffer 缓存是一个由 buffer 组成的双端链表。由函数 binit 用静态数组 buf 初始化这个链表，binit 在启动时由 main 调用。访问 buffer 缓存是通过链表，而不是 buf 数组。buffer 有两个与之相关的状态字段。字段 valid 表示是否包含该块的副本（是否从磁盘读取了数据）。字段 disk 表示缓冲区的内容已经被修改需要被重新写入磁盘。

bget 扫描 buffer 链表，寻找给定设备号和扇区号来查找缓冲区。如果存在，bget 就会获取该 buffer 的 sleep-lock。然后 bget 返回被锁定的 buffer。如果给定的扇区没有缓存的 buffer，bget 必须生成一个，可能会使用一个存放不同扇区的 buffer，它再次扫描 buffer 链表，寻找没有被使用的 buffer(b->refcnt = 0)；任何这样的 buffer 都可以使用。任何这样的 buffer 都可以使用。bget 修改 buffer 元数据，记录新的设备号和扇区号，并获得其 sleep-lock。请注意，b->valid = 0 可以确保 bread 从磁盘读取块数据，而不是错误地使用 buffer 之前的内容。如果所有 buffer 都在使用，那么太多的进程同时在执行文件相关的系统调用，bget 就会 panic。一个更好的处理方式可能是睡眠，直到有 buffer 空闲，尽管这时有可

能出现死锁。

请注意每个磁盘扇区最多只能有一个 buffer，以确保写操作对读取者可见，也因为文件系统需要使用 buffer 上的锁来进行同步。Bget 通过从第一次循环检查块是否被缓存，第二次循环来生成一个相应的 buffer（通过设置 dev、blockno 和 refcnt），在进行这两步操作时，需要一直持有 bache.lock。持有 bache.lock 会保证上面两个循环在整体上是原子的。

bget 在 bcache.lock 保护的临界区之外获取 buffer 的 sleep-lock 是安全的，因为非零的 b->refcnt 可以防止缓冲区被重新用于不同的磁盘块。sleep-lock 保护的是块的缓冲内容的读写，而 bcache.lock 保护被缓存块的信息。

bread, bwrite: 一旦 bread 读取了磁盘内容（如果需要的话）并将缓冲区返回给它的调用者，调用者就独占该 buffer，可以读取或写入数据。如果调用者修改了 buffer，它必须在释放 buffer 之前调用 bwrite 将修改后的数据写入磁盘。

brelease: 当调用者处理完一个 buffer 后，必须调用 brelease 来释放它。brelease 释放 sleep-lock，并将该 buffer 移动到链表的头部。移动 buffer 会使链表按照 buffer 最近使用的时间（最近释放）排序，链表中的第一个 buffer 是最近使用的，最后一个是最早使用的。bget 中的两个循环利用了这一点，在最坏的情况下，获取已缓存 buffer 的扫描必须处理整个链表，由于数据局部性，先检查最近使用的缓冲区（从 bcache.head 开始，通过 next 指针）将减少扫描时间。扫描选取可使用 buffer 的方法是通过从后向前扫描（通过 prev 指针）选取最近使用最少的缓冲区。

代码修改（以 bio.c 为基础）

kernel/memlayout.h

```
1
2 添加 #define MAX_USED_PHYSADDR    0x100000000// 最大物理地
    址，用于memmove转移时不出错
3 添加 #define P2V(p)                (p+KERNBASE)
```

kernel/string.c

```
1
2 添加 #include "memlayout.h"
3 修改memmove（变成对实际物理地址的修改）
4 void*
5 memmove(void *dst, const void *src, uint n)
6 {
```

```

7  const char *s;
8  char *d;
9  if(n == 0)
10     return dst;
11  s = src;
12  d = dst;
13  if ((uint64)dst <= (uint64)MAX_USED_PHYSADDR)
14     d = P2V(dst);
15  if ((uint64)src <= (uint64)MAX_USED_PHYSADDR)
16     s = P2V(src);
17  if(s < d && s + n > d){
18     s += n;
19     d += n;
20     while(n-- > 0)
21         *--d = *--s;
22  } else
23     while(n-- > 0)
24         *d++ = *s++;
25  return dst;
26 }

```

kernel/ramdisk.c (利用 memmove 简化实现)

```

1  void
2  ramdiskrw(struct buf *b, int w)
3  {
4      uint64 diskaddr = b->blockno * BSIZE;
5      char *addr = (char *)RAMDISK + diskaddr;
6      if(!w && !b->valid)
7          memmove(b->data, addr, BSIZE);
8      else if (w)
9          memmove(addr, b->data, BSIZE);
10 }

```

kernel/bio.c

```

1  virtio_disk_rw->ramdiskrw
2  riscv.h->loogarch.h

```

3.1.3 logging

日志贮存在一个固定位置，由 superblock 指定。它由一个 header 块组成，后面是一连串的更新块副本（日志块）。header 块包含一个扇区号数组，其中的每个扇区号都对应一个日志块 [1]，header 还包含日志块的数量。磁盘上 header 块中的数量要么为零，表示日志中没有事务，要么为非零，表示日志中包含一个完整的提交事务，并有指定数量的日志块。

`log_write` 是 `bwrite` 的代理。它将扇区号记录在内存中，在磁盘上的日志中使用一个槽，并自增 `buffer.refcnt` 防止该 buffer 被重用。在提交之前，块必须留在缓存中，即该缓存的副本是修改的唯一记录；在提交之后才能将其写入磁盘上的位置；该次修改必须对其他读可见。注意，当一个块在一个事务中被多次写入时，他们在日志中的槽是相同的。这种优化通常被称为 `absorption`(吸收)。例如，在一个事务中，包含多个文件的多个 inode 的磁盘块被写多次，这是常见的情况。通过将几次磁盘写吸收为一次，文件系统可以节省日志空间，并且可以获得更好的性能，因为只有一份磁盘块的副本必须写入磁盘。

`end_op` 首先递减 `log.outstanding`。如果计数为零，则通过调用 `**commit()` 来提交当前事务。

Commit 分为四个阶段：

-
- 1 1、`write_log()` 将事务中修改的每个块从 `buffer` 缓存中复制到磁盘上的日志槽中。
 - 2
 - 3 2、`write_head()` 将 `header` 块写到磁盘上，就表明已提交，为提交点，写完日志后的崩溃，会导致在重启后重新执行日志。
 - 4
 - 5 3、`install_trans` 从日志中读取每个块，并将其写到文件系统中对应的位置。
 - 6
 - 7 4、最后修改日志块计数为 0，并写入日志空间的 `header` 部分。这必须在下一个事务开始之前修改，这样崩溃就不会导致重启后的恢复使用这次的 `header` 和下次的日志块。
-

`recover_from_log` 是在 `initlog` 中调用的，而 `initlog` 是在第一个用户进程运行之前，由 `fsinit` 调用的。它读取日志头，如果日志头显示日志中包含一个已提交的事务，则会像 `end_op` 那样执行日志。

3.1.4 Inode

磁盘上的 inode 被放置磁盘的一个连续区域。每一个 inode 的大小都是一样的，所以，给定一个数字 n ，很容易找到磁盘上的第 n 个 inode。事实上，这个数字 n ，被称为 inode 号或 i-number，在实现中就是通过这个识别 inode 的。

结构体 dinode 定义了磁盘上的 inode。type 字段区分了文件、目录和特殊文件（设备）。type 为 0 表示该 inode 是空闲的。nlink 字段统计引用这个 inode 的目录项的数量，当引用数为 0 时就释放磁盘上的 inode 及其数据块。size 字段记录了文件中内容的字节数。addrs 数组记录了持有文件内容的磁盘块的块号。

内核将在使用的 inode 保存在内存中；结构体 inode 是磁盘 dinode 的拷贝。内核只有在有指针指向 inode 才会储存。ref 字段为指向 inode 的指针的数量，如果引用数量减少到零，内核就会从内存中丢弃这个 inode。iget 和 iput 函数引用和释放 inode，并修改引用计数。指向 inode 的指针可以来自文件描述符，当前工作目录，以及短暂的内核代码，如 exec。

iget() 返回的 inode 指针在调用 iput() 之前都是有效的；inode 不会被删除，指针所引用的内存也不会被另一个 inode 重新使用。**iget() 提供了对 inode 的非独占性访问，因此可以有許多指针指向同一个 inode。文件系统代码中的许多部分都依赖于 iget() 的这种行爲，既是为了保持对 inode 的长期引用（如打开的文件和当前目录），也是为了防止竞争，同时避免在操作多个 inode 的代码中出现死锁（如路径名查找）。

inode 缓存只缓存被指针指向的 inode。它的主要工作其实是同步多个进程的访问，缓存是次要的。如果一个 inode 被频繁使用，如果不被 inode 缓存保存，buffer 缓存可能会把它保存在内存中。inode 缓存是 write-through 的，这意味着缓存的 inode 被修改，就必须立即用 iupdate 把它写入磁盘。

Inode content

磁盘上的 inode，即 dinode 结构体，包含一个 size 和一个块号数组（见图）。inode 数据可以在 dinode 的 addrs 数组中找到。开始的 NDIRECT 个数据块列在数组中的前 NDIRECT 个条目中，这些块被称为直接块。接下来的 NINDIRECT 个数据块并没有列在 inode 中，而是列在叫做间接块的数据块中。addrs 数组中的最后一个条目给出了放置间接块的地址。因此，一个文件的前 12 kB (NDIRECT x BSIZE) 字节可以从 inode 中列出的块中加载，而接下来的 256 kB (NINDIRECT x BSIZE) 字节只能在查阅间接块后才能取出。对于磁盘这是一种不错的表示方式，但对客户机就有点复杂了。函数 bmap 包装了这种表示方式使得高层次的函数，如 readi 和 writei 可以更好的使用。Bmap 返回 inodeip

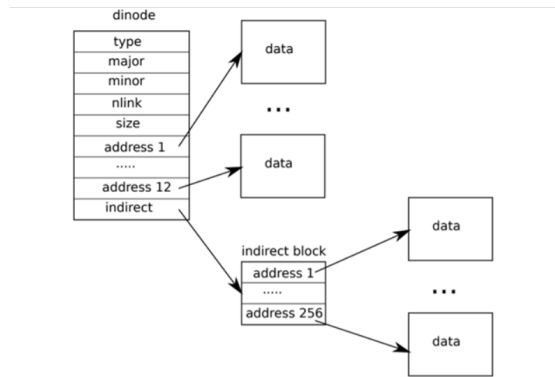


图 3.2: 文件系统结构

的第 bn 个数据块的磁盘块号。如果 ip 没有第 bn 个的数据块, $bmap$ 就会分配一个。

3.1.5 directory

目录的实现机制和文件很类似。它的 $inode$ 类型是 T_DIR , 它的数据是一个目录项的序列。每个条目是一个结构体 $dirent$, 它包含一个名称和一个 $inode$ 号。名称最多包含 $DIRSIZ(14)$ 个字符, 较短的名称以 $NULL(0)$ 结束。 $inode$ 号为 0 的目录项是空闲的。

dirlookup 在一个目录中搜索一个带有给定名称的条目。如果找到了, 它返回一个指向相应未上锁的 $inode$ 的指针, 并将 $poff$ 设置为目录中条目的字节偏移量, 以便调用者想要编辑它。如果 $dirlookup$ 找到一个对应名称的条目, 则更新 $poff$, 并返回一个通过 $iget$ 获得的未被锁定的 $inode$ 。 $Dirlookup$ 是 $iget$ 返回未锁定的 $inode$ 的原因。调用者已经锁定了 dp , 所以如果查找的是 “.”, 当前目录的别名, 在返回之前试图锁定 $inode$, 就会试图重新锁定 dp 而死锁。调用者可以先解锁 dp , 然后再锁定 ip , 保证一次只持有一个锁。

dirlink 会在当前目录 dp 中创建一个新的目录项, 通过给定的名称和 $inode$ 号。如果名称已经存在, $dirlink$ 将返回一个错误。主循环读取目录项, 寻找一个未使用的条目。当它找到一个时, 它会提前跳出循环, 并将 off 设置为该可用条目的偏移量。否则, 循环结束时, 将 off 设置为 $dp->size$ 。不管是哪种方式, $dirlink$ 都会在偏移量 off 的位置添加一个新的条目到目录中。

3.1.6 Path names

查找路径名会对每一个节点调用一次 $dirlookup$ 。 $Namei$ 解析路径并返回相应的 $inode$ 。函数 $nameiparent$ 是 $namei$ 的一个变种: 它返回相应 $inode$ 的父目

录 inode，并将最后一个元素复制到 name 中。这两个函数都通过调用 namex 来实现。

Namex 首先确定路径解析从哪里开始。如果路径以斜线开头，则从根目录开始解析；否则，从当前目录开始解析。然后它使用 skipelem 来遍历路径中的每个元素。循环的每次迭代都必须在当前 inode ip 中查找 name。迭代的开始是锁定 ip 并检查它是否是一个目录。如果不是，查找就会失败。（锁定 ip 是必要的，不是因为 ip->type 可能会改变，而是因为在 ilock 运行之前，不能保证 ip->type 已经从磁盘载入）。如果调用的是 nameiparent，而且这是最后一个路径元素，按照之前 nameiparent 的定义，循环应该提前停止，最后一个路径元素已经被复制到 name 中，所以 namex 只需要返回解锁的 ip。最后，循环使用 dirlookup 查找路径元素，并通过设置 ip = next 为下一次迭代做准备。当循环遍历完路径元素时，它返回 ip。

namex 可能需要很长的时间来完成：它可能会涉及几个磁盘操作，通过遍历路径名得到的目录的 inode 和目录块（如果它们不在 buffer 缓存中）。Xv6 经过精心设计，如果一个内核线程对 namex 的调用阻塞在磁盘 I/O 上，另一个内核线程查找不同的路径名可以同时进行。Namex 分别锁定路径中的每个目录，这样不同目录的查找就可以并行进行。

3.1.7 File descriptor

系统给每个进程提供了自己的打开文件表，或者说文件描述符表，就像我们在第一章中看到的那样。每个打开的文件由一个结构体 file 表示，它包装 inode 或管道，也包含一个 I/O 偏移量。每次调用 open 都会创建一个新的打开文件（一个新的结构体 file），如果多个进程独立打开同一个文件，那么不同的 file 实例会有不同的 I/O 偏移量。另一方面，一个打开的文件（同一个结构文件）可以在一个进程的文件表中出现多次，也可以在多个进程的文件表中出现。如果一个进程使用 open 打开文件，然后使用 dup 创建别名，或者使用 fork 与子进程共享文件，就会出现这种情况。引用计数可以跟踪特定打开文件的引用数量。一个文件的打开方式可以为读，写，或者读写。通过 readable 和 writable 来指明。

系统中所有打开的文件都保存在一个全局文件表中，即 ftable。文件表的功能有：分配文件 (filealloc)、创建重复引用 (fileup)、释放引用 (fileclose)、读写数据 (fileearead 和 filewrite)。

Filealloc 扫描文件表，寻找一个未引用的文件 (f->ref == 0)，并返回一个新的引用；**fileup** 增加引用计数；**fileclose** 减少引用计数。当一个文件的引用数达到 0 时，fileclose 会根据类型释放底层的管道或 inode。

函数 `filestat`、`fileread` 和 `filewrite` 实现了对文件的统计、读和写操作。`Filestat` 只允许对 inodes 进行操作，并调用 `stat`。Fileread 和 filewrite 首先检查打开模式是否允许该操作，然后再调用管道或 inode 的相关实现。如果文件代表一个 inode，fileread 和 filewrite 使用 I/O 偏移量作为本次操作的偏移量，然后前移偏移量。

Chapter 4

调度

4.1 基本概念

任何操作系统运行的进程数量都可能超过计算机的 CPU 数量，因此需要制定一个方案，在各进程之间分时共享 CPU。

4.1.1 Context switching

从一个用户进程切换到另一个用户进程所涉及的步骤：用户-内核的切换（通过系统调用或中断）到旧进程的内核线程，上下文（context）切换到当前 CPU 的调度器线程，上下文（context）切换到新进程的内核线程，以及 trap 返回到用户级进程。调度器在每个 CPU 上有一个专门的线程（保存了寄存器和栈），因为调度器在旧进程的内核栈上执行是不安全的：因为其他核心可能会唤醒该进程并运行它，而在两个不同的核心上使用相同的栈将是一场灾难。在本节中，我们将研究在内核线程和调度线程之间切换的机制。

从一个线程切换到另一个线程，需要保存旧线程的 CPU 寄存器，并恢复新线程之前保存的寄存器；栈指针和 pc 被保存和恢复，意味着 CPU 将切换栈和正在执行的代码。

swtch 执行内核线程切换的保存和恢复。**swtch** 并不直接知道线程，它只是保存和恢复寄存器组，称为上下文 (context)。当一个进程要放弃 CPU 的时候，进程的内核线程会调用 **swtch** 保存自己的上下文并返回到调度器上下文。每个上下文都包含在一个结构体 context 中，它本身包含在进程的结构体 proc 或 CPU 的结构体 cpu 中。**Swtch** 有两个参数：struct context old 和 struct context new。它将当前的寄存器保存在 old 中，从 new 中加载寄存器，然后返回。。

Swtch 只保存 callee-saved 寄存器，caller-saved 寄存器由调用的 C 代码保

存在堆栈上 (如果需要)。Swch 知道 struct context 中每个寄存器字段的偏移量。它不保存 pc。相反, swch 保存了 ra 寄存器 [1], 它保存了 swch 应该返回的地址。现在, swch 从新的上下文中恢复寄存器, 新的上下文中保存着前一次 swch 所保存的寄存器值。当 swch 返回时, 它返回到被恢复的 ra 寄存器所指向的指令, 也就是新线程之前调用 swch 的指令。此外, 它还会返回新线程的堆栈。

4.1.2 Scheduling

scheduler 运行了一个简单的循环: 找到一个可以运行进程, 运行它, 直到它让出 CPU, 一直重复。调度器在进程表上循环寻找一个可运行的进程, 即 `p->state == RUNNABLE` 的进程。一旦找到这样的进程, 它就会设置 CPU 当前进程变量 `c->proc` 指向该进程, 将该进程标记为 `RUNNING`, 然后调用 `swch` 开始运行它。

你可以这样理解调度代码结构, 它执行一组关于进程的不变量, 并且每当这些不变量不正确时, 就持有 `p->lock`。一个不变量是, 如果一个进程正在运行, 那么定时中断导致的 `yield` 必须能够安全的让他让出 `cpu`; 这意味着 CPU 寄存器必须持有该进程的寄存器值 (即 `swch` 没有将它们移到上下文中), 并且 `c->proc` 必须指向该进程。另一个不变量是, 如果一个进程是 `RUNNABLE` 的, 那么对于一个空闲的 CPU 调度器来说, 运行它必须是安全的; 这意味着 (1) `p->context` 必须拥有进程的寄存器 (即它们实际上并不在真实的寄存器中), (2) 没有 CPU 在进程的内核栈上执行, (3) 也没有 CPU 的 `c->proc` 指向该进程。请注意, 当 `p->lock` 被持有时, 这些属性往往不正确。

维护上述不变量是 `xv6` 经常在一个线程中获取 `p->lock`, 然后在另一个线程中释放它的原因 (例如在 `yield` 中获取, 在 `scheduler` 中释放)。一旦 `yield` 开始修改一个正在运行的进程的状态, 使其成为 `RUNNABLE`, 锁必须一直保持, 直到不变量被恢复: 最早正确的释放点是在调度器 (运行在自己的堆栈上) 清除 `c->proc` 之后。同样, 一旦调度器开始将一个 `RUNNABLE` 进程转换为 `RUNNING`, 锁就不能被释放, 直到内核线程完成运行 (在 `swch` 之后, 例如在 `yield` 中)。

4.1.3 代码修改

根据 loongarch 结构, 修改 `kernel/trampoline.S`

```
1
2 # save the user registers in TRAPFRAME
3     st.d    $ra, $a0, 40
4     st.d    $tp, $a0, 48
```

```

5      st.d    $sp, $a0, 56
6      st.d    $a1, $a0, 72
7      st.d    $a2, $a0, 80
8      st.d    $a3, $a0, 88
9      st.d    $a4, $a0, 96
10     st.d    $a5, $a0, 104
11     st.d    $a6, $a0, 112
12     st.d    $a7, $a0, 120
13     st.d    $t0, $a0, 128
14     st.d    $t1, $a0, 136
15     st.d    $t2, $a0, 144
16     st.d    $t3, $a0, 152
17     st.d    $t4, $a0, 160
18     st.d    $t5, $a0, 168
19     st.d    $t6, $a0, 176
20     st.d    $t7, $a0, 184
21     st.d    $t8, $a0, 192
22     st.d    $r21, $a0, 200
23     st.d    $fp, $a0, 208
24     st.d    $s0, $a0, 216
25     st.d    $s1, $a0, 224
26     st.d    $s2, $a0, 232
27     st.d    $s3, $a0, 240
28     st.d    $s4, $a0, 248
29     st.d    $s5, $a0, 256
30     st.d    $s6, $a0, 264
31     st.d    $s7, $a0, 272
32     st.d    $s8, $a0, 280

```

```

1
2      # restore all but a0 from TRAPFRAME
3      ld.d    $ra, $a0, 40
4      ld.d    $tp, $a0, 48
5      ld.d    $sp, $a0, 56
6      ld.d    $a1, $a0, 72
7      ld.d    $a2, $a0, 80
8      ld.d    $a3, $a0, 88

```

```

9      ld.d      $a4, $a0, 96
10     ld.d      $a5, $a0, 104
11     ld.d      $a6, $a0, 112
12     ld.d      $a7, $a0, 120
13     ld.d      $t0, $a0, 128
14     ld.d      $t1, $a0, 136
15     ld.d      $t2, $a0, 144
16     ld.d      $t3, $a0, 152
17     ld.d      $t4, $a0, 160
18     ld.d      $t5, $a0, 168
19     ld.d      $t6, $a0, 176
20     ld.d      $t7, $a0, 184
21     ld.d      $t8, $a0, 192
22     ld.d      $r21, $a0, 200
23     ld.d      $fp, $a0, 208
24     ld.d      $s0, $a0, 216
25     ld.d      $s1, $a0, 224
26     ld.d      $s2, $a0, 232
27     ld.d      $s3, $a0, 240
28     ld.d      $s4, $a0, 248
29     ld.d      $s5, $a0, 256
30     ld.d      $s6, $a0, 264
31     ld.d      $s7, $a0, 272
32     ld.d      $s8, $a0, 280
33
34     ld.d      $a0, $a0, 64

```

根据 loongarch 结构，修改 kernel/swtch.S

```

1     st.d      $ra, $a0, 0
2     st.d      $sp, $a0, 8
3     st.d      $s0, $a0, 16
4     st.d      $s1, $a0, 24
5     st.d      $s2, $a0, 32
6     st.d      $s3, $a0, 40
7     st.d      $s4, $a0, 48
8     st.d      $s5, $a0, 56
9     st.d      $s6, $a0, 64

```

```
10      st.d    $s7, $a0, 72
11      st.d    $s8, $a0, 80
12
13      ld.d    $ra, $a1, 0
14      ld.d    $sp, $a1, 8
15      ld.d    $s0, $a1, 16
16      ld.d    $s1, $a1, 24
17      ld.d    $s2, $a1, 32
18      ld.d    $s3, $a1, 40
19      ld.d    $s4, $a1, 48
20      ld.d    $s5, $a1, 56
21      ld.d    $s6, $a1, 64
22      ld.d    $s7, $a1, 72
23      ld.d    $s8, $a1, 80
24
25      jirl    $zero, $ra, 0
```
