ucore-mips移植报告

2014011 郭昉泽 2014011 马强 2014011319 王倩

移植以操作系统课程lab8为基础，以ucore-thumips为参考进行修改移植。

基本

基本MIPS32S指令集与x86指令集的区别

- 所有指令都是32位的
- 算术/逻辑运算指令为3操作数指令格式，两个源操作数和一个目标操作数只能是寄存器操作数
- **32个通用寄存器**
  - 相关修改：./user/libs/initCode.S
- 对内存的访问只能通过装入(load)和存储(store)指令进行。算术/逻辑运算指令不能直接访问内存
- **寻址方式**：基址寄存器+16位有符号偏移量
- 数据在内存中存储时必须按边界对齐
- 一些特殊指令
  - cache
  - eret: Return from exception
  - MFC0/MTC0: move from/to coprocessor register
  - TLBWI: TLB maintenance
  - SYSCALL: system call exception

mips与x86在其他方面的区别

- **没有**专门的堆栈支持
  - 相关修改：./user/libs/initCode.S
- **子程序调用**通过"跳转与链接指令"进行， 将返回地址保存在一个寄存器中($31)。
- **中断**发生时，处理器只负责跳到预定义的地址处

mips寄存器

- 通用寄存器

寄存器编号	助记符	用法
0	zero	永远为0
1	at	用做汇编器的暂时变量
2-3	v0, v1	子函数调用返回结果
4-7	a0-a3	子函数调用的参数
8-15	t0-t7	临时寄存器。在子函数中使用时不需要保存与恢复
24-25	t8-t9	
16-23	s0-s7	保存寄存器。在子函数中使用这些寄存器时，子函数必须保存和恢复这些寄存器的原值
26,27	k0,k1	通常被中断或异常处理程序使用，用来保存一些系统参数
28	gp	全局指针。一些运行系统维护这个指针来更方便的存取static和extern变量
29	sp	堆栈指针
30	s8/fp	第9个保存寄存器/帧指针
31	ra	子函数的返回地址

- CP0的寄存器（中断、mmu的控制都在这部分）

名称	编号	说明	实现的域
SR	12	状态寄存器	IP,KSU,EXL,IE
Cause	13	中断原因	IP,ExcCode
EPC	14	中断返回地址	全部
BadVAddr	8	异常虚拟地址	全部
EntryHi	10	TLB，存储VPN	VPN2
EntryLo0	2	TLB，存储PFN等	PFN,D,V
EntryLo1	3	TLB，存储PFN等	PFN,D,V
Index	0	TLB，写入编号	全部（由于TLB只有16项，因而高28位会忽略）
Count	9	记时寄存器	全部
Compare	11	比较寄存器	全部
EBase	15	中断基址寄存器	ExceptionBase

- LOHI寄存器
- 特权等级：当KSU=0或EXL=1时位于内核态
- 指令/访存的虚拟地址必须位于[0x00000000,0x7FFFFFFF]

主要修改模块1：启动

重写boot汇编代码

修改：./boot/bootasm.S
boot流程：

- 进入读flash模式

- 检查magic
- 获取基本信息elfheader
- 挨扇区读取elf
- 结束后跳到kernel

参考代码：

```
load_elf:
  #get ready to read flash
  la $t0, (FLASH_START+FLASH_SIZE - 8)
  la $t1, 0x00FF
  sw $t1, 0($t0)

  #addr of elfheader, s0
  la $s0, FLASH_START
  #e_magic
  LOAD_WORD_I($t1, 0)
  la $t0, ELF_MAGIC
  beq $t0, $t1, 1f
  nop
  b bad
  nop
1:
  #e_phoff
  LOAD_WORD_I($s1, 28)
  #e_phnum
  LOAD_WORD_I($s2, 44)
  andi $s2, $s2, 0xFFFF

  #e_entry
  LOAD_WORD_I($s3, 24)

next_sec:
  #s1, addr proghdr
  #s4, p_va
  LOAD_WORD_R($s4, 8, $s1)
  #s5, p_filesz
  LOAD_WORD_R($s5, 16, $s1)
  #s6, p_offset
  LOAD_WORD_R($s6, 4, $s1)

  beq  $s4, $zero, 3f
  nop
  beq  $s5, $zero, 3f
  nop

#copy from file_base+p_offset to p_va
copy_sec:
  LOAD_WORD_R($t0, 0, $s6)
  sw $t0, 0($s4)
  addiu $s6, $s6, 4
  addiu $s4, $s4, 4
  addiu $s5, $s5, -4
  bgtz  $s5, copy_sec
  nop

3:
  addiu $s1, $s1, 32
  addiu $s2, $s2, -1
  bgtz  $s2, next_sec
  nop

done:
#jump to kernel
  jr $s3
  nop
  b .
  nop
```

修改说明：

- 没有了x86的修改A20过程，没有从实模式切换到保护模式
- 也省去了用bootmain来读取elf的过程，直接用汇编读取

init

修改位置：./kern/init/
修改文件：entry.S init.c
修改内容：

- entry.S用mips32重写，设置CP0和堆栈

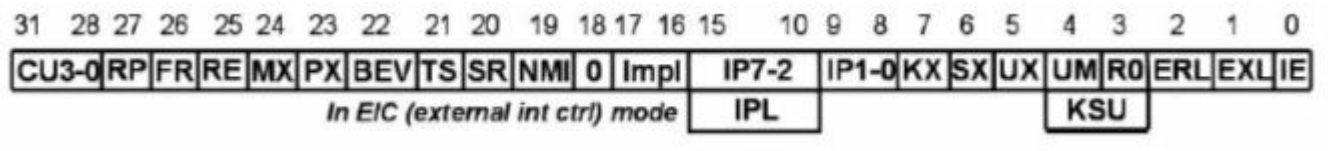
```
reset:
  /* from u-boot */
#ifdef MACH_QEMU
        /* Clear watch registers */
        mtc0    $zero, CP0_WATCHLO
        mtc0    $zero, CP0_WATCHHI
#endif

        /* WP(Watch Pending), SW0/1 should be cleared */
        mtc0    $zero, CP0_CAUSE
  /* clear SR(ERL), which is 1 after reset */
        mtc0    $zero, CP0_STATUS

  jal 1f
  nop
  .word _gp
1:
  lw $gp, 0($ra)
  la $sp, bootstacktop
#setup ram exception
  la   $t0, __exception_vector
#TODO
  mtc0 $t0, $15, 1

  mfc0 $t0, CP0_STATUS
  li   $t1, ~ST0_BEV
  and  $t0, $t0, $t1
  mtc0 $t0, CP0_STATUS

#zero bss
  la $t0, edata
  la $t1, end

2:
  sw $zero, 0($t0)
  addiu $t0, $t0, 4
  slt $t3, $t0, $t1
  bgtz $t3, 2b
  nop

  addiu $sp, $sp, -16
  la $t9, kern_init
  jal $t9
```

- init.c把lab1相关的内容去掉了，然后swap算法用tlb方法替代，有相关的更改。

主要修改模块2：中断

基本说明

- 一旦CPU判断中断条件成立，就进行中断操作
- 一旦执行ERET指令，就进行中断返回操作
- 中断号：7-时钟；4-串口；6-PS2键盘
- 总线的Int是中断总线，6条 **中断流程**
- 将当前PC写入CP0的EPC寄存器；
- 将当前访存的虚拟地址写入CP0的BadVAddr寄存器；
- 将异常原因写入CP0的Cause寄存器；
- 将SR寄存器的EXL置‘1’；(SR寄存器的结构如下)



- 如果是TLB缺失异常，将BadVAddr的VPN写入EntryHi寄存器；
- 将中断入口地址写入PC。中断入口地址是Base+0x180；

中断返回

- 将SR寄存器的EXL置'0'；
- 将EPC寄存器写入PC

异常流程

- ERL在reset后要clear
- 处理内部的异常要设置EX和KSU
- user与kernel切换时，kernel用trapframe来存registers

时钟中断(7)

- 通过Count/Compare寄存器实现。
- 当Count寄存器的值等于Compare寄存器时发出时钟中断
- 写入Compare寄存器会清除中断

硬件中断

- EXL=0
- IE=1
- IM的对应位=1
- IP的对应位=1（来自中断总线）

修改流程

- 修改文件：trap.c
- 主要修改函数：void trap_dispatch(struct trapframe *tf)
- 修改内容：trap的类型和原来的不太一样，其中
  - 中断单独用handler处理

```
static void interrupt_handler(struct trapframe *tf)
{
  extern clock_int_handler(void*);
  extern serial_int_handler(void*);
  int i;
  for(i=0;i<8;i++){
    if(tf->tf_cause & (1<<(CAUSEB_IP+i))){
      switch(i){
        case TIMER0_IRQ:
          clock_int_handler(NULL);
          break;
        case COM1_IRQ:
          serial_int_handler(NULL);
          break;
        default:
          print_trapframe(tf);
          panic("Unknown interrupt!");
      }
    }
  }

}
```

  - tlb缺失单独handler处理（具体操作写在./kern/include/thumips_tlb.h里）

```
static inline void write_one_tlb(int index, unsigned int pagemask, unsigned int hi, unsigned int low0, unsigned int low1)
{
        write_c0_entrylo0(low0);
        write_c0_pagemask(pagemask);
        write_c0_entrylo1(low1);
        write_c0_entryhi(hi);
```

```
        write_c0_index(index);
        tlb_write_indexed();
}

static inline void tlb_replace_random(unsigned int pagemask, unsigned int hi, unsigned int low0, unsigned int low1)
{
        write_c0_entrylo0(low0);
        write_c0_pagemask(pagemask);
        write_c0_entrylo1(low1);
        write_c0_entryhi(hi);
  tlb_write_random();
}

#define PTE2TLBLOW(x) (((((uint32_t)(*(x))-KERNBASE)>> 12)<<6)|THUMIPS_TLB_ENTRYL_V|THUMIPS_TLB_ENTRYL_D|(2<<3))
static inline uint32_t pte2tlblow(pte_t pte)
{
  uint32_t t = (((uint32_t)pte - KERNBASE ) >> 12)<<6;
  if(!ptep_present(&pte))
    return 0;
  t |= THUMIPS_TLB_ENTRYL_V;
  /* always ignore ASID */
  t |= THUMIPS_TLB_ENTRYL_G;
  t |= (2<<3);
  if(ptep_s_write(&pte))
    t |= THUMIPS_TLB_ENTRYL_D;
  return t;
}

static inline void tlb_refill(uint32_t badaddr, pte_t *pte)
{
  if(!pte)
    return ;
  if(badaddr & (1<<12))
    pte--;
  tlb_replace_random(0, badaddr & THUMIPS_TLB_ENTRYH_VPN2_MASK,
      pte2tlblow(*pte), pte2tlblow(*(pte+1)));
}

void tlb_invalidate_all();
#endif
```

  - trapframe和regs的打印都要调整
  - 没有了idt相关的内容

- 修改文件：vectors.S
  - 修改内容：写中断处理程序的地址记录，都从0x180写入，对应的处理流程在exception.S

- 修改文件：exception.S(原trapentry.S)
  - 修改内容：ramExcHandle_general为处理程序，进行基本的用户态核心态不同的操作后进入common_exception，保存现场，处理异常（mips_trap），恢复现场，返回。需要注意的地方前面都有说。

主要修改模块3：存储访问和存储管理

基本说明

- 没有MMU但有programmable TLB单元
- kseg1和kseg0直接映射，其他的用TLB miss exception 来模拟MMU

TLB

- 低12位页内，13位选择，高19位是vpn
- 每一项TLB有两个pfn和一个vpn
- 结构（D1/D2用于只读异常的判断，传给上层；V1/V2...）

修改说明

- pmm.c作为框架改动不大，简化了一些地方
  - 在init时的check系列函数里要执行tlb_invalidate_all()
  - 删去换页的地方
  - 由于mips的tlb的项与虚页号是直接匹配的，所以pmm_init()里少了map相关的内容。
  - 用current_pgdir来代替cr2记录基址

- kmalloc.c(旧代码几乎没有复用)
  - slab用了cache来加速

```
typedef struct kmem_cache_s kmem_cache_t;
struct kmem_cache_s {
    list_entry_t slabs_full;     // list for fully allocated slabs
    list_entry_t slabs_notfull;  // list for not-fully allocated slabs

    size_t objsize;              // the fixed size of obj
    size_t objsize_shift;
    size_t num;                  // number of objs per slab
    size_t offset;               // this first obj's offset in slab
    bool off_slab;               // the control part of slab in slab or not.
    size_t page_order;
    kmem_cache_t *slab_cachep;
};
#define MIN_SIZE_ORDER          5           // 32
#define MAX_SIZE_ORDER          17          // 128k
#define SLAB_CACHE_NUM          (MAX_SIZE_ORDER - MIN_SIZE_ORDER + 1)

static kmem_cache_t slab_cache[SLAB_CACHE_NUM];
static void init_kmem_cache(kmem_cache_t *cachep, size_t objsize);
static void check_slab(void);
```

```
// init_kmem_cache - initial a slab_cache cachep according to the obj with the size = objsize
static void
init_kmem_cache(kmem_cache_t *cachep, size_t objsize) {}
static void *kmem_cache_alloc(kmem_cache_t *cachep);
#define slab_bufctl(slabp)

// kmem_cache_slabmgmt - get the address of a slab according to page
static slab_t *
kmem_cache_slabmgmt(kmem_cache_t *cachep, struct Page *page) {}

#define SET_PAGE_CACHE(page, cachep)

#define SET_PAGE_SLAB(page, slabp)

// kmem_cache_grow - allocate a new slab by calling alloc_pages
static bool
kmem_cache_grow(kmem_cache_t *cachep) {}

// kmem_cache_alloc_one - allocate a obj in a slab
static void *
kmem_cache_alloc_one(kmem_cache_t *cachep, slab_t *slabp) {}

// kmem_cache_alloc - call kmem_cache_alloc_one function to allocate a obj
static void *
kmem_cache_alloc(kmem_cache_t *cachep) {}
```

- vmm.c
  - 除了没有swap以外没有太大变化

主要修改模块4：进程管理

说明

context

- s0-s8给子函数用
- gp是全局指针
- ra是中断前pc/子函数返回地址
- sp是堆栈指针（切换环境的时候要用）

修改

修改：./process/entry.S
修改说明：用mips重写了调用进程的过程

```
kernel_thread_entry:
  addiu $sp, $sp, -16
```

```
  jal $a1
  nop
  move $a0, $v0
  la  $t0, do_exit
  jal $t0
  nop
  /* never here */
```

修改：./process/switch.S
修改说明：用mips重写了切换进程时寄存器的行为

```
/* save the registers */
sw      sp, 44(a0)
...
nop

/* restore the registers */
lw      s0, 0(a1)
...
nop                     /* delay slot for load */

/* return. */
j ra
nop
```

修改：./process/proc.c
修改说明：

- 由于trapframe的变化而导致一些细节上的变化

```
tf.tf_regs.reg_r[MIPS_REG_A1] = (uint32_t)fn;
tf.tf_regs.reg_r[MIPS_REG_V0] = 0;
tf.tf_status = read_c0_status();
tf.tf_status &= ~ST0_KSU;
tf.tf_status |= ST0_IE;
tf.tf_status |= ST0_EXL;
```

- 有部分函数涉及汇编语言的重新写

```
asm volatile(

  "la $v0, %1;\n" /* syscall no. */
  "move $a0, %2;\n"
  "move $a1, %3;\n"
  "move $a2, %4;\n"
  "move $a3, %5;\n"
  "syscall;\n"
  "nop;\n"
  "move %0, $v0;\n"
  : "=r"(ret)
  : "i"(SYSCALL_BASE+SYS_exec), "r"(name), "r"(argc), "r"(argv), "r"(argc)
  : "a0", "a1", "a2", "a3", "v0"
);
```

- lab4和lab6相关的内容不需要
- 其他部分改动不大

修改：./kern/schedule/default_sched.c
修改说明：把lab6所用的方法去掉即可

修改：./user/libs/syscall.c
背景知识：

- a0-a3是参数
- v0是syscall的代码，也存着return value

修改说明：

- 按照指导课件修改了inline syscall函数

修改：./user/libs/initCode.S
修改说明：不需要参数相关的操作，也不需要栈相关的操作，直接修改sp模拟栈

```
la $gp, _gp
addiu $sp, $sp, -16
jal umain
nop
```

修改：./user/*

- 除了原来的还增加了助教给的新的测试程序

修改： ./dev/intr.c

- intr.c的实现方法不同，mips是对c0进行修改

主要修改模块5：综合

修改/include/*

- 新增了thumips.h与thumips_tlb.h，mips_vm.h，asm/
- 其他部分改动不大

修改： ./kern/include/thumips.h
这里是一些会用到的基本操作，需要重写

```
static inline uint8_t
inb(uint32_t port) {
    uint8_t data = *((volatile uint8_t*) port);
    return data;
}
static inline uint32_t
inw(uint32_t port) {
    uint32_t data = *((volatile uintptr_t *) port);
    return data;
}

static inline void
outb(uint32_t port, uint8_t data) {
    *((volatile uint8_t*) port) = data;
}
static inline void
outw(uint32_t port, uint32_t data) {
    *((volatile uintptr_t *) port) = data;
}
#define __read_reg(source)

static inline unsigned int __mulu10(unsigned int n){
  return (n<<3)+(n<<1);
}

static inline unsigned int __divu10(unsigned int n) {}
```

修改： libs/*

- 新增printfmt.c
- 新增stdio.h
- 新增string.c/h
- 删去了lab6相关的内容