

使用说明

此工具用于在 `rCore` panic 时进行堆栈回溯，打印函数调用路径。目前只在 ch9 分支上进行测试，从 ch6 开启文件系统后均可以使用此工具。

如何获取函数信息

在栈回溯时，需要查询函数信息，而这些函数信息主要包含于可执行文件中。具体的细节可以在 [elf 文件函数信息](#) 这里查看，这里给出主要的查找过程

```
graph LR
a(遍历ELF的section) --> B(根据section类型找到.symtab段)
B --> c(遍历.symtab的Symbol Table Entry)
c --> d(找到entry类型为函数的项)
d --> e(获取函数名称)
d --> f(获取函数起始地址)
d --> g(获取函数范围)
e --> h(解析函数名称)
```

由于 `rust` 会对函数名称进行重整，类似于 C++，因此需要使用相应工具进行解析才能转为可读的名称。同时，汇编文件中的函数可能不会被上述过程收集到。

栈回溯分类

主要有两种堆栈回溯方式，一种是使用 `sp` 和 `fp` 指针进行回溯，如下图所示，在这种方式下，每当函数进行开辟栈帧操作后，就会保存 `ra` `tp` 的值，然后令 `fp` 指向当前的栈顶，



在这种情况下，在进行栈回溯时，首先根据 `fp` 寄存器指向的地址，取出保存在函数栈中 `ra` 和 `fp` 寄存器的数据，`ra` 的值是函数返回地址，`fp` 的值是上一级函数栈的栈顶地址，根据 `ra` 的值到收集的函数信息中查找此地址是否位于某个函数的范围，如果是，则记录函数信息，然后根据 `fp` 回到上一级函数，继续读取 `ra` 和 `fp` 的值，指导无法找到对应的函数区间。

第二种回溯方式是由于某些编译器不会利用 `fp` 生成上述的代码，从而需要额外的手段进行解析，有的会使用 ELF 文件中的 `.eh_frame` 段内容，由于这些方式比较复杂，因此本工具暂不使用此方法。

本工具回溯方法

本工具根据一般函数生成形式，比如rust生成的一段 `risc-v` 代码如下

```
0000000080210412 <my_trace>:
 80210412: 7149          addi    sp,sp,-368
 80210414: f686          sd      ra,360(sp)
 80210416: f2a2          sd      s0,352(sp)
 80210418: eea6          sd      s1,344(sp)
 8021041a: eaca          sd      s2,336(sp)
 8021041c: e6ce          sd      s3,328(sp)
 8021041e: e2d2          sd      s4,320(sp)
 80210420: fe56          sd      s5,312(sp)
 80210422: fa5a          sd      s6,304(sp)
```

可以看到，函数的前两条指令是开辟栈空间和保存 `ra` 的指令，因此这里一个简单的想法就是通过读取函数的第一条指令和第二条指令，获取到开辟的栈空间大小以及 `ra` 存储的位置，这里 `ra` 一般就是存储在栈顶，读取第二条指令主要是确保这条指令是保存 `ra` 的指令。再使用汇编指令读取当前的 `sp` 值，就可以得到下面的回溯方式：

读取函数第一条指令和第二条指令获得栈大小size

栈底: sp

栈顶: sp + size

ra : m[sp+size-8]

寻找ra所在函数

将找到的函数设置为当前函数

再次重复上述过程

因此主要工作在于如何解析函数的第一条和第二条指令，通过查询risc-v手册可以找到各条指令的格式，比如 addi 指令的格式

addi rd, rs1, immediate

$x[rd] = x[rs1] + \text{sext(immediate)}$

加立即数(*Add Immediate*). I-type, RV32I and RV64I.

把符号位扩展的立即数加到寄存器 $x[rs1]$ 上，结果写入 $x[rd]$ 。忽略算术溢出。

压缩形式: c.li rd, imm; c.addi rd, imm; c.addi16sp imm; c.addi4spn rd, imm

31	20 19	15 14	12 11	7 6	0
	immediate[11:0]	rs1	000	rd	0010011

读取第一条指令并按照上面的格式解析出立即数部分就可以得到栈大小，但由于risc-v的编译器会做某些优化，将 addi 指令使用压缩指令表示，而压缩指令一般是两字节格式，比如 c.addi 指令的格式如下：

c.addi rd, imm

$x[rd] = x[rd] + \text{sext(imm)}$

加立即数 (*Add Immediate*). RV32IC and RV64IC.

扩展形式为 addi rd, rd, imm.

15	13	12	11	7 6	2 1	0
000	imm[5]		rd	imm[4:0]	01	

因此需要根据压缩指令和未压缩的指令共同判断第一条指令是否未开辟栈空间的指令和栈空间大小。同理，判断第二条指令也需要如上的工作。

具体的实现请查看源代码。

使用方法

本工具以一个库的形式提供，需要传入的参数为OS的可执行文件。

```

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    if let Some(location) = info.location() {
        println!(
            "[kernel] Panicked at {}:{} {}",
            location.file(),
            location.line(),
            info.message().unwrap()
        );
    } else {
        println!("[kernel] Panicked: {}", info.message().unwrap());
    }
    unsafe {
        backtrace();
    }
    shutdown(255)
}

use stack_trace::{Trace};
unsafe fn backtrace() {
    let mut os_name:Vec<&str> = Vec::new();
    let all_file = ROOT_INODE.ls();
    all_file.iter().for_each(|x| {
        if x.contains("os") {
            os_name.push(x);
        }
    });
    os_name.sort(); //由于内核文件较大，会被分成多个文件
    let mut data = Vec::new();
    os_name.iter().for_each(|name| {
        let mut file = open_file(*name,OpenFlags::RDONLY).unwrap();
        let d = file.read_all();
        trace!("name: {} {}", name, d.len());
        data.extend_from_slice(d.as_slice());
    });
    let mut trace = Trace::new();
    trace.init(data.as_slice()); //初始化，传入内核可执行文件
    let road = trace.trace(); //收集函数调用信息
    road.iter().for_each(|s|{
        println!("{} {}", s);
    });
}

```

目前的内核无法传递参数，为了在内核中读取本身的ELF文件，需要在编译后将ELF文件与应用程序文件一样打包在fs.img中，为了不将代码写死，这里在easy-fs-fuse添加了一个参数：

```

.arg(
    Arg::with_name("kernel")
        .short("k")
        .long("kernel")
        .takes_value(true)
        .help("Kernel source dir(with backslash)"),
)

```

同时在os的Makefile文件也需要修改相应的参数：

```
@cd .. /easy-fs-fuse && cargo run --release -- -s .. /user/src/bin/ -t  
.. /user/target/riscv64gc-unknown-none-elf/release/ -k .. /os/target/riscv64gc-  
unknown-none-elf/release/
```

然后只需要在easy-fs-fuse将内核文件写入fs-img, 由于内核可执行文件高达16MB, 并且os中文件系统最多支持8MB大小的文件, 因此在打包时需要将文件进行切分, 并且设置fs-img大小为32MB

```
let block_file : Arc<BlockFile> = Arc::new( data: BlockFile(Mutex::new( t {  
    let f : File = OpenOptions::new()  
        .read(true) : &mut OpenOptions  
        .write(true) : &mut OpenOptions  
        .create(true) : &mut OpenOptions  
        .open( path: format!("{}{}", target_path, "fs.img"))?  
    f.set_len(32 * 2048 * 512).unwrap();  
    f  
})));  
// 16*2MiB, at most 4095*2 files  
let efs : Arc<Mutex<EasyFileSystem>> = EasyFileSystem::create( block_device: block_file, total_blocks: 32 * 2048, inode_bitmap_blocks: 1);  
  
let kernel_path = matches.value_of("kernel").unwrap();  
println!("kernel path = {}{}", kernel_path, "os");  
let mut file = File::open(format!("{}{}", kernel_path, "os")).unwrap();  
let mut all_data: Vec<u8> = Vec::new();  
file.read_to_end(&mut all_data).unwrap();  
println!("{}", all_data.len());  
//如果数据大于8MB,将数据切分  
let mut data_vec: Vec<Vec<u8>> = Vec::new();  
let mut data_vec_len = 0;  
while data_vec_len < all_data.len() {  
    let mut data_vec_tmp: Vec<u8> = Vec::new();  
    data_vec_tmp.extend_from_slice(&all_data[data_vec_len..(data_vec_len +  
8 * 1024 * 1024).min(all_data.len())]);  
    data_vec.push(data_vec_tmp);  
    data_vec_len += 8 * 1024 * 1024;  
}  
for i in 0..data_vec.len() {  
    let inode = root_inode.create(format!("os{}", i).as_str()).unwrap();  
    inode.write_at(0, data_vec[i].as_slice());  
}
```

完成上述步骤, 就可以在内核中读取本身的ELF文件并传入栈回溯库了(内核读取这种大文件有点慢)

目前在内核中使用此功能可以达到的效果如下所示

```
initproc
infloop
sync_sem
pipe_large_test
filetest_simple
fantastic_text
os0
os1
*****
[kernel] Panicked at src/test.rs:14 test panic
0x80210438 (+16) stack_trace::trace::Trace::trace
0x80206db8 (+1116) os::lang_items::backtrace
0x80206c1e (+196) rust_begin_unwind
0x8022243c (+44) core::panicking::panic_fmt
0x80205880 (+104) os::test::test_stack_trace
0x8020ae1e (+428) rust_main
make: *** [Makefile:109: run-inner] 错误 1
```

```
os0
os1
*****
[kernel] Panicked at src/task/process.rs:144 [KERNEL_EXEC]
0x8021dc5a (+16) stack_trace::trace::Trace::trace
0x80209f52 (+1116) os::lang_items::backtrace
0x80209db8 (+196) rust_begin_unwind
0x802301e2 (+44) core::panicking::panic_fmt
0x8020e56a (+320) os::task::process::ProcessControlBlock::exec
0x8021b4ae (+510) os::syscall::process::sys_exec
0x80217c50 (+168) trap_handler
make: *** [Makefile:109: run-inner] 错误 1
```

注意事项

目前的实现仍然比较简陋，且限制较大，由于在出错时需要读取文件内容，此时如果发生任务调度可能会造成死锁的问题，但这是在ch9会发生的现象，如果在前面的章节中，读取文件内容是阻塞式的不会发生任务调度，因此应该不会造成死锁问题。

为了解决这个问题，可以在开始进入用户态之前就读取内核数据，后面如果发生错误，就不需要读取文件发生死锁。上面的代码修改为：

```

lazy_static!{
    static ref KERNEL_DATA: UPIIntrFreeCell<Vec<u8>> =
unsafe{UPIIntrFreeCell::new(Vec::new())};
}

pub fn init_kernel_data(){
    let mut os_name:Vec<&str> = Vec::new();
    let all_file = ROOT_INODE.ls();
    all_file.iter().for_each(|x| {
        if x.contains("os") {
            os_name.push(x);
        }
    });
    os_name.sort();
    os_name.iter().for_each(|name| {
        let mut file = open_file(*name,OpenFlags::RDONLY).unwrap();
        let d = file.read_all();
        trace!("name: {} {}",name,d.len());
        KERNEL_DATA.exclusive_access().extend_from_slice(d.as_slice());
    });
}

unsafe fn backtrace() {
    let mut trace = Trace::new();
    trace.init(KERNEL_DATA.exclusive_access().as_slice());
    let road = trace.trace();
    road.iter().for_each(|s|{
        println!("{}",s);
    });
}

```

在main函数中，在开始进入用户程序之前调用 `init_kernel_data()` 即可。

改进

- 在编译前获取函数信息并与内核一同链接 --> 适用于所有章节的栈回溯方法
- 使用 `.eh_frame` 进行栈回溯而不是读取函数的前两条指令

环境配置

在进行实验之前，需要安装一些基本的工具和搭建实验环境。本实验需要在linux操作系统上进行。下面的操作均在wsl2下完成。

Qemu安装

目前qemu主分支上的可能与龙芯仓库中不一致，因此需要注意安装的方式。

方式1:源码构建

```
git clone https://github.com/foxsen/qemu.git  
git checkout loongarch  
cd qemu  
mkdir build  
cd ./build/  
../configure --target-list=loongarch64-softmmu --disable-werror (调试版本加--enable-debug) make -j4  
make install
```

注意在编译完成后可能仍然无法使用提示缺少相关的动态库，这时可自行上网查看相应的安装方法，对于缺失的库，其安装方式大都是相同的，只是在名字上有一点差异。

方式2: 使用项目目录下的可执行文件

在项目根目录下存在一个 `qemu-loongarch-runenv` 目录，里面包含了可执行文件 `qemu-system-loongarch64`，

要成功使用此文件，也需要安装多个依赖库，可以根据名称提示安装相应的库。

上述两个方式均需要使用 `bios` 文件作为启动器，此文件也位于目录 `qemu-loongarch-runenv` 中。

GCC工具链

由于在项目中需要使用 `gdb` 以及 `objdump`、`readelf` 工具，这些工具均包含在此工具链中。

1. 从 `qemu-loongarch-runenv` 下载对应gcc交叉编译工具链
2. 解压并添加环境变量

Rust安装

目前官方仍然没有添加 LoongArch64 平台，后续应该会很快支持，因此这里只能暂时使用老师提供的一个版本。[百度网盘地址链接](#) 提取码：fuoi

里面包含有如何安装的细节

linker文件

链接脚本的主要目的是描述如何将输入文件中的各个section（节）映射到输出文件中，并控制输出文件的内存布局。链接器总是使用链接脚本，如果不主动修改，链接器将使用一个默认的链接脚本，这个脚本被编译进了链接器可执行文件中。

链接器将输入文件组合成一个输出文件。输出文件和每个输入文件都采用一种特殊的数据格式，称为目标文件格式。每个文件称为一个目标文件。输出文件通常称为可执行文件，有时也仍然会称为目标文件。每个目标文件都有一个段（section）列表。有时把输入文件的段称作输入段，类似的，输出文件的段称作输出段。目标文件中的每个段都有名称和大小。大多数段还具有关联的数据块，称为段内容。一个段可能被标记为可加载(loadable)，这意味着在运行输出文件时，段内容需要先加载到内存中。一个没有内容的段是可分配的，这意味着应该在内存中预留一个区域，但是这里不需要加载任何东西（在某些情况下，该内存必须清零）。既不可装载也不可分配的部分通常包含某种调试信息。

使用下面的命令查看文件中的section信息：

```
objdump -h x.out
```

对于 loongArch64 使用 loongarch64-unknown-linux-gnu-objdump

每个目标文件还具有一个符号列表，称为符号表。符号可以是定义的也可以是未定义的。每个符号都有一个名称，每个定义的符号都有一个地址，以及其他信息。如果将C或C++程序编译到目标文件中，则将会将所有定义过的函数和全局变量以及静态变量作为已定义符号。输入文件中引用的每个未定义函数或全局变量都将成为未定义符号。

使用linux命令 nm 查看符号信息。

链接脚本是一种特殊的文本文件。文本描述是一系列的命令，每个命令都是一个关键字，可能后面还跟有一个参数，或者一个符号的赋值。使用分号分割命令，空格通常被忽略。

一个最简单的链接脚本文件如下：

```
SECTIONS
{
    . = 0x10000;
    .text : { *(.text) }
    . = 0x80000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

最简单的链接脚本只有一个命令： SECTIONS，这个命令用来描述输出文件的内存布局。

. 符号是location counter，用于指定section的地址，SECTIONS命令开始时，其值为0，可以显式设置，如果没有设置，则按照section大小自动增长。.text 定义一个output section。后面跟一

个冒号，现在可以省略。大括号用来指定input sections，*是通配符，匹配任何文件名。表达式'*(.text)'表示输入文件的所有 .text section。链接器保证output section满足对齐要求。

程序执行的第一条指令叫做entry point。ENTRY命令用于设置入口点。链接器按照下面的顺序寻找入口点。

1. -e 命令行选项。
2. ENTRY(symbol)链接脚本命令。
3. 目标指定的符号，通常是start。
4. .text的第一个字节。
5. 0

在操作系统实验中一般使用第二种方法指定。

PHDRS 命令用于创建程序头，链接器缺省或创建合适的程序头。

链接脚本也可以定义变量，这时只会生成一个 symbol 项，并不会分配内存。在目标文件内定义的符号也可以在链接脚本内被赋值。符号定义是全局的。每个符号都对应了一个地址。使用 nm 命令时可以看到定义的符号，在内核代码中需要查看各个段的内存范围需要在链接脚本提供相应的符号。

PROVIDE 关键字于定义这类符号：在目标文件内被引用，但没有在任何目标文件内被定义的 符号

```
SECTIONS
{
    .text :
    {
        *(.text)
        _etext = .;
        PROVIDE(etext = .);
    }
}
```

当目标文件内引用了 etext 符号，却没有定义它时，etext 符号对应的地址被定义为 .text section 之后的第一个字节的地址。

ALIGN(align) 命令常用来指定对齐地址。

Rust裸机环境配置

在默认情况下，Rust 尝试适配当前的系统环境，编译可执行程序。为了描述不同的环境，Rust 使用一个称为**目标三元组** (target triple) 的字符串。要查看当前系统的**目标三元组**，我们可以运行 `rustc --version --verbose`。Rust 编译器尝试为当前系统的三元组编译，并假定底层有一个类似于 Windows 或 Linux 的操作系统提供 C 语言运行环境——然而这将导致链接器错误。所以，为了避免这个错误，需要另选一个底层没有操作系统的运行环境，这样的环境被称为**裸机环境**。在 `risc-v` 平台上，rust 原生就有支持相应的 `riscv64gc-unknown-none-elf` 裸机平台。但对于 `loongArch64` 平台来说，并没有相应的支持，目前只含有 `loongarch64-unknown-linux-gnu`、`loongarch64-unknown-linux-musl` 两个支持，而这两个三元组都默认底层有 linux 系统支持，因此想要编译裸机代码，就需要去掉标准库支持。

通常我们需要在项目根目录下创建 `.cargo/config` 文件，并写入相应的配置。在当前项目下的内容如下：

```
[build]
target = "loongarch64-unknown-linux-gnu"
[target.loongarch64-unknown-linux-gnu]
linker = "loongarch64-unknown-linux-gnu-gcc"
```

`target` 指定了编译的目标平台，`linker` 指定了所用的链接脚本，虽然这里指定了配置，但后续介绍的 `build.rs` 会修改相关的规则才能编译裸机代码。

这里出现的一个问题是若像 rCore 官方一样在 `config` 文件指定链接脚本的话似乎并不会起作用，需要 `build.rs` 的帮助。因此上面的 `linker` 命令也是多余命令

Rust的build.rs文件

在项目下存在一个 `build.rs` 文件。一些项目希望编译第三方的非 Rust 代码，例如 C 依赖库；一些希望链接本地或者基于源码构建的 C 依赖库；还有一些项目需要功能性的工具，例如在构建之间执行一些代码生成的工作等。对于这些目标，Cargo 提供了自定义构建脚本的方式，来帮助用户更好的解决类似的问题。

只需在项目的根目录下添加一个 `build.rs` 文件即可。这样一来，Cargo 就会先编译和执行该构建脚本，然后再去构建整个项目

构建脚本如果会产出文件，那么这些文件需要放在统一的目录中，该目录可以通过 `OUT_DIR` 环境变量来指定，构建脚本不应该修改该目录之外的任何文件！

构建脚本可以通过 `println!` 输出内容跟 Cargo 进行通信：Cargo 会将每一行带有 `cargo:` 前缀的输出解析为一条指令，其它的输出内容会自动被忽略。其中一些我们需要的命令包括

1. cargo:rerun-if-changed=PATH 表示如果 PATH 的内容如果发生更改，则需要重新运行 build.rs
2. cargo:rustc-link-search=[KIND=]PATH 告知 Cargo 通过 -L 将一个目录添加到依赖库的搜索路径中
3. cargo:rustc-link-arg=FLAG - 将自定义的 flags 传给 linker，用于后续的基准性能测试 benchmark、可执行文件 binary,、cdylib 包、示例和测试

构建脚本打印到标准输出 `stdout` 的所有内容将保存在文件

`target/debug/build/<pkg>/output` 中(具体的位置可能取决于你的配置)，`stderr` 的输出内容也将保存在同一个目录中。

为了编译 loongArch64 的裸机代码，`build.rs` 需要执行的操作如下：

```
let outdir = env::var("OUT_DIR").unwrap();
let link_script = Path::new(&outdir).join("link.lds");
let mut script = File::create(&link_script).unwrap();
script.write_all(include_bytes!("linker.ld")).unwrap();
println!("cargo:rustc-link-arg=-T{}", &link_script.display());
println!("cargo:rustc-link-arg=-nostdlib"); //关闭gcc的默认链接
```

第一步是将默认的链接脚本指定为自定义脚本

第二步是将链接脚本传递给编译器

第三步是关闭标准库

在实验前期，仍然会使用到 `build.rs` 文件来完成一些生成代码工作。

参考链接

[build.rs](#)

[config](#)

Rust汇编

在编写操作系统的进程中，需要使用汇编代码完成部分工作。Rust作为一种系统编程语言，同样也支持在代码中插入汇编代码。

嵌入汇编代码方式

目前有两种嵌入汇编代码的方式，在实践过程中均会使用到

- 直接编写.asm文件，使用 `global_asm!(include_str!("head.S"))`；宏将汇编文件与源代码一起编译
- 在函数中使用 `asm!()` 插入汇编代码

一个简单的内联汇编代码如下

```
use std::arch::asm;
let a: i64 = 1;
let b: i64;
unsafe {
    asm!("mov {0}, {1}", out(reg) b, in(reg) a);
}
println!("{}", b);
```

上面的 `out` 操作数表示输出，`in` 表示操作数输入，`b` 为目标变量，`reg` 寄存器类则是让Rust编译器自动分配一个寄存器，当寄存器的值被更新后会再读取其中的值到变量 `b` 中（也就是写到变量 `b` 所在的栈地址）

在 loongArch64 平台上，内联汇编的支持可能没有 x86 或者 risc-v 一样好，比如在 risc-v 平台上下面的代码是正确的：

```
unsafe {
    asm!("mv {0}, {1}", out(t0) b, in(t1) a);
}
```

但是类似的代码在 loongArch64 上不能运行

```
unsafe {
    asm!("move {0}, {1}", out("t0") b, in("t1") a);
}
unsafe {
    asm!("mv {0}, {1}", out("$t0") b, in("$t1") a);
}
```

因此在后续的用户态 `syscall` 中不能按照 rCore 中一样直接在函数中嵌入内联汇编，需要单独编写一个汇编文件。

参考资料

一些更详细的内联汇编说明和 loongArch 汇编代码编写方式可在如下网址查看

[内联汇编](#)

[loongArch汇编](#)

第一章

在第一章中，介绍了如何进行移除标准库以及如何进入使用 Rust 编写内核代码的世界。

- 启动过程
- UART串口打印

启动过程

计算机启动过程

无论采用何种指令系统的处理器，复位后的第一条指令都会从一个预先定义的特定地址取回。处理器的执行就从这条指令开始。处理器的启动过程，实际上就是一个特定程序的执行过程。这个程序我们称之为固件，又称为 BIOS (Basic Input Output System, 基本输入输出系统)。对于 LoongArch，处理器复位后的第一条指令将固定从地址 0x1C000000 的位置获取。这个地址需要对应一个能够给处理器核提供指令的设备，这个设备以前是各种 ROM，现在通常是闪存 (Flash)。从获取第一条指令开始，计算机系统的启动过程也就开始了。在 risc-v 体系结构上，通常这个地址是 0x80200000。在启动过程中，计算机需要对包括处理器核、内存、外设等在内的各个部分分别进行初始化，再对必要的外设进行驱动管理。

RISC-V 架构中，存在着定义于操作系统之下的运行环境。这个运行环境不仅将引导启动 RISC-V 下的操作系统，还将常驻后台，为操作系统提供一系列二进制接口，以便其获取和操作硬件信息。RISC-V 给出了此类环境和二进制接口的规范，称为“操作系统二进制接口”，即“SBI”。RustSbi 作为其中一种实现，在当前 rCore 中使用。其位于 risc-v 定义的 M 态下，比操作系统的级别更高，对机器有更大的权限。机器上电时，SBI 将配置环境，准备设备树，最终将引导启动操作系统。操作系统需要访问硬件或者特殊的功能，这时候就需要通过 ecall 指令陷入 M 层的 SBI 运行时，由 SBI 完成这些功能再提供。

在 LoongArch 或者 x86 这些架构下，通常上述工作由 BIOS 完成，现在一般是 UEFI。UEFI 和 UEFI 提供了整个主板，包括主板上外插的设备的软件抽象，通过探测，枚举，找到系统所有的硬件信息，再通过几组详细定义好的接口，把这些信息抽象封装后传递给操作系统，这些信息包括 SMBIOS，ACPI 表等，通过这层映射，操作系统就能做到在不修改的前提下直接运行在新的硬件上。通常来说，UEFI 不会像 SBI 一样一直位于后台运行，在内核代码中，一般只会去读取 UEFI 提供的信息而不主动调用其实现的一些功能。

UEFI/BIOS

为了在 Qemu 上启动 LoongArch 的机器，需要一个 UEFI 启动器，因此在 qemu-loongarch-runenv 目录下提供了此文件。

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 vmlinux 看到）抹去高 32 位使用。比如 vmlinux 链接的地址是 0x9000000001034804，实际 bios 跳转的地址将是 0x1034804，代码装载的位置也是物理内存 0x1034804。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。

- 内核启动入口代码需要做两件事：（参见 arch/loongarch/kernel/head.S）
 1. 设置一个直接地址映射窗口（参见 loongarch 体系结构手册，5.2.1 节），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-xxxxxxxx 和

0x9000xxxx-xxxxxxxx地址抹去最高的8和9为其物理地址，前者用于uncache访问(即不通过高速缓存去load/store)，后者用于cache访问。

2. 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。

在linux源代码中可以得到入口代码如下所示：

```
SYM_CODE_START(kernel_entry)                                # kernel entry point
    la.abs          t0, 0f
    jirl            zero, t0, 0
0:
    la              t0, __bss_start      # clear .bss
    st.d            zero, t0, 0
    la              t1, __bss_stop - LONGSIZE
1:
    addi.d          t0, t0, LONGSIZE
    st.d            zero, t0, 0
    bne             t0, t1, 1b

    #设置直接地址映射窗口
    li.d            t0, CSR_DMW0_INIT      # UC, PLV0, 0x8000 xxxx xxxx
xxxx
    csrwr           t0, LOONGARCH_CSR_DMWIN0
    li.d            t0, CSR_DMW1_INIT      # CA, PLV0, 0x9000 xxxx xxxx
xxxx
    csrwr           t0, LOONGARCH_CSR_DMWIN1
    #开启页表
    li.w            t0, 0xb0            # PLV=0, IE=0, PG=1
    csrwr           t0, LOONGARCH_CSR_CRMD
    li.w            t0, 0x04            # PLV=0, PIE=1, PWE=0
    csrwr           t0, LOONGARCH_CSR_PRMD
    li.w            t0, 0x00            # FPE=0, SXE=0, ASXE=0, BTE=0
    csrwr           t0, LOONGARCH_CSR_EUEN

    #设置栈空间
    PTR_LI          sp, (_THREAD_SIZE - 32 - PT_SIZE)
    PTR_ADDU        sp, sp, tp
    set_saved_sp   sp, t0, t1
    PTR_ADDIU       sp, sp, -4 * SZREG     # init stack pointer

    #跳转到内核入口
    bl              start_kernel

SYM_CODE_END(kernel_entry)
```

乍一看怎么才第一章就已经开始这么难了，但实际上上述的代码我们可以在后续章节中再使用，这里我们将会使用另外一种更简单的方式。

在内存使用上，BIOS实现了虚拟地址和物理地址相等的一个映射。为了简单起见，第一章的内核利用了这个映射，跳过了常规的汇编初始化代码。在实际的内核代码中，内核将会接管物理内存和虚拟内存，不能一直依赖BIOS建立的映射，当然也要注意使用的内存不会破坏BIOS用于传递参数的区域。因此这里可以直接开始编写rust代码。

```
#[no_mangle]
pub extern "C" fn main(){
    INFO!("{}", FLAG);
    color_output_test();
    panic!();
}
```

在linker文件中，指定了入口为main函数，因此这里关闭了rust的函数名重整，这样才能正确链接到符号。其中 `color_output_test` 与 `rCore` 中打印链接脚本中提供的各个段的地址代码相同。

链接脚本如下所示：

```

OUTPUT_ARCH( "loongarch" )
ENTRY( main)

SECTIONS
{
    . = 0x00100000;
    text_start = .;
    .text : {
        *(.text .text.*)
        PROVIDE(etext = .);
        . = ALIGN(0x1000);
    }
    text_end = .;
    . = ALIGN(0x1000);
    rodata_start = .;
    .rodata : {
        . = ALIGN(16);
        *(.srodata .srodata.*)
        . = ALIGN(16);
        *(.rodata .rodata.*)
    }
    . = ALIGN(0x1000);
    rodata_end = .;
    data_start = .;
    .data : {
        . = ALIGN(16);
        *(.sdata .sdata.*)
    }
    . = ALIGN(16);
    *(.data .data.*)
}

    . = ALIGN(0x1000);
    data_end = .;
    bss_start = .;

    .bss : {
        sbss = .;
    }
    . = ALIGN(16);
    *(.sbss .sbss.*)
    . = ALIGN(16);
    *(.bss .bss.*)
}
    . = ALIGN(0x1000);
    bss_end = .;

    ekernel = .;
    PROVIDE(end = .);
}

```

其指定了内核的入口地址位于0x00100000，由上述介绍可知，BIOS会将内核代码直接加载到物理地址0x00100000处，然后跳转到此处开始运行，这里没有像linux一样进行窗口配置，一方面是因为BIOS已经为我们设置了这个窗口，并且BIOS会提供虚拟地址与物理地址之间的恒等映射，因此为了不带来更多的汇编代码配置，这里就直接使用了BIOS提供的便利。

UART串口

通用异步收发传输器（Universal Asynchronous Receiver/Transmitter，通常称为UART）是一种异步收发传输器，是电脑硬件的一部分，将数据透过串列通信进行传输。

UART 控制器具有以下特性：

- 全双工异步数据接收/发送
- 可编程的数据格式
- 16 位可编程时钟计数器
- 支持接收超时检测
- 带仲裁的多中断系统
- 仅工作在 FIFO 方式
- 在寄存器与功能上兼容 NS16550A

在UART上追加同步方式的串行信号变换电路的产品，被称为USART。

在实验初期，由于没有涉及到显示设备，因此如果想要知道系统运行过程中发生的事情就需要我们手动将其打印出来，而这个时候能用的就只有 `uart` 串口设备，通过此外设，可以将程序的输出打印到屏幕上。

通常各个厂商的 `uart` 都会兼容NS16550,而此设备只需要几个寄存器进行控制，根据不同架构，访问这些寄存器有两种方式，一种是通过端口映射的I/O访问，这种情况一般需要特殊的I/O指令，另一种方式是使用MMIO，这种情况下没有特殊指令，使用一般的访存指令即可。

UART硬件受一个内部时钟信号控制。该时钟信号是数据传输率的倍频，典型是比特率的8或16倍。接收器在每个时钟脉冲时测试接收到的信号状态是否为开始比特。如果开始比特的低电平持续传输1个比特所需时间的一半以上，则认为开始了一个数据帧的传输；否则，则忽略此脉冲信号。到了下一个比特时间后，线路状态被采样并送入移位寄存器。约定的表示一个字符的所有数据比特(典型为5至8个比特) 接收后，移位寄存器可被接收系统使用。UART将设置一个标记指出新数据可用，并产生一个处理器中断请求主机处理器取走接收到的数据。

在 `loongArch` 结构下，IO 地址空间与内存地址空间统一编址。处理器上运行的指令使用虚拟地址，虚拟地址通过地址映射规则与物理地址相关联。基本的虚拟地址属性首先区分为经缓存（Cache）与不经缓存（Uncache）两种。对于内存操作，现代高性能通用处理器都采用 Cache 方式进行访问，以提升访存性能。对于存储器来说，在 Cache 中进行缓存是没有问题的，因为存储器所存储的内容不会自行修改。但是对于 IO 设备来说，因为其寄存器状态是随着工作状态的变化而变化的，如果缓存在 Cache 中，那么处理器核将无法得到状态的更新，所以一般情况下不能对 IO 地址空间进行 Cache 访问，需要使用 Uncache 访问。使用 Uncache 访问对 IO 进行操作还有另一个作用，就是可以严格控制读写的访问顺序，不会因为预取类的操作导致寄存器状态的丢失。

在 `rCore` 中，串口的访问依靠了SBI进行实现，在 `loongArch` 下，则仍然需要编写少量代码，由于在UEFI中已经对串口设备进行了初始化设置工作，因此我们没有必要再次对其进行再次设置。通常，`uart`设备的初始化过程如下所示：

```
LEAF(initserial)
# 加载串口设备地址
li a0, GS3_UART_BASE
#线路控制寄存器，写入0x80（128）表示后续的寄存器访问为分频寄存器访问
li t1, 128
sb.b t1, a0, 3
# 配置串口波特率分频，当串口控制器输入频率为33MHz，将串口通讯速率设置在115200
# 时，分频方式为33,000,000 / 16 / 0x12 = 114583。由于串口通信有固定的起始格式，
# 能够容忍传输两端一定的速率差异，只要将传输两端的速率保持在一定的范围之内就可
# 以保证传输的正确性
li t1, 0x12
sb.b t1, a0, 0
li t1, 0x0
sb.b t1, a0, 1
# 设置传输字符宽度为8，同时设置后续访问常规寄存器
li t1, 3
sb.b t1, a0, 3
# 不使用中断模式
li t1, 0
sb.b t1, a0, 1
li t1, 71
sb t1, a0, 2
jirl ra
nop
END(initserial)
```

串口设备使用相同的地址映射了两套功能完全不同的寄存器，通过线路控制寄存器的最高位（就是串口寄存器中偏移为3的寄存器的最高位）进行切换。因为其中一套寄存器主要用于串口波特率的设置，只需要在初始化时进行访问，在正常工作状态下完全不用再次读写，所以能够将其访问地址与另一套正常工作用的寄存器相复用来节省地址空间。

在初始化时，代码中先将0x3偏移寄存器的最高位设置为1，以访问分频设置寄存器，按照与连接设备协商好的波特率和字符宽度，将初始化信息写入配置寄存器中。然后退出分频寄存器的访问模式，进入正常工作模式。在使用时，串口的对端是一个同样的串口，两个串口的发送端和接收端分别对连，通过双向的字符通信来实现被调试机的字符输出和字符输入功能。

在内核中如果想要使用串口进行输出，需要完成下面的代码：

```

pub fn put(&mut self, c: u8) {
    let mut ptr = self.base_address as *mut u8;
    loop {
        unsafe {
            let c = ptr.add(5).read_volatile();
            if c & (1<<5)!=0{
                break;
            }
        }
    }
    ptr = self.base_address as *mut u8;
    unsafe {
        ptr.add(0).write_volatile(c);
    }
}

pub fn get(&mut self) -> Option<u8> {
    let ptr = self.base_address as *mut u8;
    unsafe {
        if ptr.add(5).read_volatile() & 1 == 0 {
            // The DR bit is 0, meaning no data
            None
        } else {
            // The DR bit is 1, meaning data!
            Some(ptr.add(0).read_volatile())
        }
    }
}

```

在打印字符函数中，需要偏移地址为0x5的线路状态寄存器，检查FIFO空标志，非空时等待，否则就写入。在读取函数中则相反。

在qemu模拟的 loongarch 平台上，uart的基地址为 0x1fe001e0.

Trait和macro

本小节主要介绍rust的宏机制，在上一节中我们实现了如何在操作系统中使用串口设备，但是这依然很原始，因为每次只能打印一个字符，因此我们需要实现类似于 `println!` 一样可以格式化打印的功能。

首先可以简单了解一下 `trait`，`trait` 告诉 Rust 编译器某个特定类型拥有可能与其他类型共享的功能。可以通过 `trait` 以一种抽象的方式定义共享的行为。可以使用 `trait bounds` 指定泛型是任何拥有特定行为的类型。一个类型的行为由其可供调用的方法构成。如果可以对不同类型调用相同的方法的话，这些类型就可以共享相同的行为了。`trait` 定义是一种将方法签名组合起来的方法，目的是定义一个实现某些目的所必需的行为的集合。从上面的定义可以看到其与其它编程语言中的接口非常类似。下面是使用C++和rust分别使用接口和 `trait` 实现打印一个字符串的功能：

```
class Display{
    public:
        virtual void display() = 0;
}
class Screen:public Display{
    void display(){
        std::cout << "hello" << std::endl;
    }
}
```

```
pub trait Display{
    fn display();
}
struct Screen{};
impl Display for Screen{
    fn display(){
        println!("hello");
    }
}
```

孤儿规则: 实现 `trait` 时需要注意的一个限制是，只有当至少一个 `trait` 或者要实现 `trait` 的类型位于 `crate` 的本地作用域时，才能为该类型实现 `trait`。不能为外部类型实现外部 `trait`。

`trait` 可以提供默认实现，如果不将其进行覆写，则使用的是默认实现，除了提供接口抽象的能力，在加上泛型机制后，`trait` 的能力就会更加强大，比如当我们在函数传递一个参数时，希望这个参数可以调用某个特定的方法，这个时候函数的声明可能如下：

```
pub fn notify(item: &impl Display) {
    item::display();
}
```

对于 `item` 参数，我们指定了 `impl` 关键字和 `trait` 名称，而不是具体的类型。该参数支持任何实现了指定 `trait` 的类型。注意这里使用了引用，而不是直接使用 `impl Display`，因为 rust 需要在编译器知道类型的大小，而 `impl Display` 的大小未知，因此需要使用引用将其转为指针大小。

这个函数声明虽然没有显示出泛型，然后实际上背后就是泛型，其真正的函数定义如下：

```
pub fn notify<T:Display>(item: &T) {  
    item::display();  
}
```

其中T就是泛型参数，这在rust称为Trait Bound语法，这意味着传入的类型需要实现相应的trait，可以在一个类型上添加多个trait，从而添加更多的限制。trait还可以约束trait，如下：

```
trait Learning {}  
trait Teaching: Learning {}
```

如果一个类型需要实现Teaching，则其需要先实现Learning。使用 trait bound 还可以有条件地为类型实现方法：

```
struct Pair<T> {  
    x: T,  
    y: T,  
}  
  
impl<T: Display + PartialOrd> Pair<T> {  
    fn cmp_display(&self) {  
        if self.x >= self.y {  
            println!("The largest member is x = {}", self.x);  
        } else {  
            println!("The largest member is y = {}", self.y);  
        }  
    }  
}
```

只有那些为 T 类型实现了 `PartialOrd` trait (来允许比较) 和 `Display` trait (来启用打印) 的 `Pair<T>` 才会实现 `cmp_display` 方法：

在大致了解了trait的作用后就可以知道在 `println!` 的实现中需要先自定义一个结构体为其实现 `Write` trait了。此trait包含了三个函数，在将其中的 `write_str` 函数实现后，另外两个函数就不需要实现了，这时因为另外两个函数只依赖此函数。

当然实现了上述trait之后，确实可以进行打印字符串等等操作了，但是我们仍然无法进行格式化的输出，这就到了实现 `println!` 宏的部分了。

macro

宏 (Macro) 指的是 Rust 中一系列的功能：使用 `macro_rules!` 的 **声明 (Declarative)** 宏，和三种 **过程 (Procedural)** 宏：

- 自定义 `#[derive]` 宏在结构体和枚举上指定通过 `derive` 属性添加的代码
- 类属性 (Attribute-like) 宏定义可用于任意项的自定义属性

- 类函数宏看起来像函数不过作用于作为参数传递的 token

这里我们主要关注的是声明宏，其它三种过程宏中自定义宏我们会经常使用，其主要用来为自定义类型实现一些基本的 trait，比如 debug。

声明宏允许我们编写一些类似 Rust `match` 表达式的代码，`match` 表达式是控制结构，其接收一个表达式，与表达式的结果进行模式匹配，然后根据模式匹配执行相关代码。宏也将一个值和包含相关代码的模式进行比较；此种情况下，该值是传递给宏的 Rust 源代码字面值，模式用于和前面提到的源代码字面值进行比较，每个模式的相关代码会替换传递给宏的代码。所有这一切都发生在编译时。

一个简单的rust宏如下：

```
macro_rules! create_function {
    // 此宏接受一个 `ident` 指示符表示的参数，并创建一个名为 `$func_name` 的函数。
    // `ident` 指示符用于变量名或函数名
    ($func_name:ident) => (
        fn $func_name() {
            // `stringify!` 宏把 `ident` 转换成字符串。
            println!("You called {:?}()", stringify!($func_name))
        }
    )
}
```

宏的参数使用一个美元符号 `$` 作为前缀，并使用一个指示符 (designator) 来注明类，一些常见的指示符类型如下：

- `block`
- `expr` 用于表达式
- `ident` 用于变量名或函数名
- `item`
- `pat` (模式 *pattern*)
- `path`
- `stmt` (语句 *statement*)
- `tt` (标记树 *token tree*)
- `ty` (类型 *type*)

宏在参数列表中可以使用 `+` 来表示一个参数可能出现一次或多次，使用 `*` 来表示该参数可能出现零次或多次。这里我们直接分析 `print` 和 `println!` 的官方实现：

```
#[macro_export]
macro_rules! print {
    ($($arg:tt)*) => {{ $crate::io::_print($crate::format_args!($($arg)*)); }};
}
```

`format_args`宏从传递给`_print`的参数中构建一个`fmt::Arguments`类型，再调用自定义的`_print`函数打印，由于我们已经为结构体实现了`Write` trait 的`write_str`，因此可以直接调用`write_fmt`函数。

```
macro_rules! println {
    () => {
        $crate::print!("\\n")
    };
    ($($arg:tt)*) => {{
        $crate::io::_print($crate::format_args_nl!($($arg)*));
    }};
}
```

`println` 的实现中增添了空参数的匹配项，因此当参数为空时会直接打印换行符，而`format_args_nl`只是`format_args`的特殊版，其在最后加入了换行符。

第二章

第二章的任务主要是完成批处理系统，第一次编写应用程序并完成特权级的切换。为了完成这个任务，需要了解关于loongarch的部分如下：

- 例外类型
- 中断类型
- 特权指令
- 寄存器

特权级架构

龙芯架构定义了 4 个运行特权等级 (Privilege LeVel, 简称 PLV)，分别是 PLV0~PLV3。应用软件应运行在 PLV1~PLV3 这三个非特权的等级上，从而与运行在 PLV0 级上的操作系统等系统软件隔离开。应用软件具体运行在哪个特权等级上是由系统软件在运行时决定的，应用软件对此无法确切感知。龙芯架构下，应用软件通常运行在 PLV3 级上。与 risc-v 不同，loongarch 架构下没有所谓的 M 态。

刚开机时，CPU 初始化为操作系统核心态对应的运行模式，执行引导程序加载操作系统。操作系统做完一系列初始化后，控制 CPU 切换到操作系统用户态对应的运行模式去执行应用程序。应用程序执行过程中，如果出现用户态对应的运行模式无法处理的事件，则 CPU 会通过异常或中断回到核心态对应的运行模式，执行操作系统提供的服务程序。操作系统完成处理后再控制 CPU 返回用户态对应的运行模式，继续运行原来的应用程序或者调度另一个应用程序。在 LoongArch 指令系统中，CPU 当前所处的运行模式由当前模式信息控制状态寄存器 (CSR.CRMD) 的 PLV 域的值确定，其值为 0 ~ 3 分别表示 CPU 正处于 PLV0 ~ PLV3 四种运行模式。

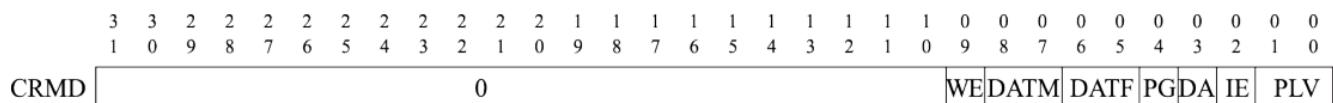


图 3.1: LoongArch 当前模式信息控制状态寄存器格式

当前我们只需要关注其低两位，与特权级相关的还有 CSR.PRMD 的寄存器

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。

在后面会介绍如何使用这两个寄存器完成特权级切换。

特权指令

所有特权指令仅在 PLV0 特权等级下才能访问。仅有一个例外情况，当 CSR.MISC 中的 RPCNTL1/RPCNTL2/RPCNTL3 配置为 1 时，可以在 PLV1/PLV2/PLV3 特权等级下执行 CSRRD 指令读取性能计数器。在本实验中并不涉及到性能计数器，因此可以不用关注。

在实验中，现阶段常用到的特权指令有：

```
csrrd rd, csr_num  
csrwr rd, csr_num  
ertn  
idle
```

CSRRD 指令将指定 CSR 的值写入到通用寄存器 rd 中。CSRWR 指令将通用寄存器 rd 中的旧值写入到指定 CSR 中，同时将指定 CSR 的旧值更新到通用寄存器 rd 中。所有 CSR 寄存器的位宽要么是 32 位，要么与架构中的 GR 等宽，因此 CSR 访问指令不区分位宽。在 LA32 架构下，所有 CSR 自然都是 32 位宽。在 LA64 架构下，定义中宽度固定为 32 位的 CSR 总是符号扩展后写入到通用寄存器 rd 中的。当 CSR 访问指令访问一个架构中未定义或硬件未实现的 CSR 时，读动作可返回任意值（推荐返回全 0 值），写动作不修改处理器的任何软件可见状态。

ertn 指令用于 trap 上下文切换的处理返回。执行 IDLE 指令后，处理器核将停止取指进入等待状态，直至其被中断唤醒或被复位。从停止状态被中断唤醒后，处理器核执行的第一条指令是 IDLE 之后的那一条指令。

在后续实验中，会使用与 TLB 和 IO 相关的特权指令，这些后续会进行介绍。

例外

异常与中断是一种打断正常的软件执行流，切换到专门的处理函数的机制。它在各种运行模式的转换中起到关键的纽带作用。比如用户态代码执行过程中，当出现对特权空间的访问，或者访问了虚实地址映射表未定义的地址，或者需要调用操作系统服务等情况时，CPU 通过发出异常来切换到核心态，进入操作系统定义的服务函数。操作系统完成处理后，返回发生异常的代码并同时切换到用户态。通常会将中断也视为一种特殊的异常，不过中断是异步的而普通异常是同步发生的，从来源看，异常可以有以下几种：

- 外部事件：来自 CPU 核外部的事件，通常是中断
- 指令执行中的错误：执行中的指令的操作码或操作数不符合要求，例如不存在的指令、除法除以 0、地址不对齐、用户态下调用核心态专有指令或非法地址空间访问等
- 数据完整性问题：当使用 ECC 等硬件校验方式的存储器发生校验错误时，会产生异常。这个功能可以被关闭。一般不会涉及到这个处理
- 地址转换异常：在存储管理单元需要对一个内存页进行地址转换，而硬件转换表中没有有效的转换对应项可用时，会产生地址转换异常。这部分会在开启页表后进行介绍。
- 系统调用和陷入：由专有指令产生，其目的是产生操作系统可识别的异常，用于在保护模式下调用核心态的相关操作。这个是本节关注的重点。

- 浮点运算错误

loongarch平台上主要的异常如下所示:

表 3.2: LoongArch 指令系统的异常一览表

异常代号	异常编号		异常说明	所属异常类别
	Ecode	Esubcode		
PIL	0x1		load 操作页无效异常	地址转换异常
PIS	0x2		store 操作页无效异常	地址转换异常
PIF	0x3		取指操作页无效异常	地址转换异常

异常代号	异常编号		异常说明	所属异常类别
	Ecode	Esubcode		
PME	0x4		页修改异常	地址转换异常
PNR	0x5		页不可读异常	地址转换异常
PNX	0x6		页不可执行异常	地址转换异常
PPI	0x7		页权限等级不合规异常	地址转换异常
ADEF	0x8	0x0	取指地址错异常	指令执行中的错误
ADEM		0x1	访存指令地址错异常	指令执行中的错误
ALE	0x9		地址非对齐异常	指令执行中的错误
BCE	0xA		边界约束检查错异常	指令执行中的错误
SYS	0xB		系统调用异常	系统调用和陷入
BRK	0xC		断点异常	系统调用和陷入
INE	0xD		指令不存在异常	指令执行中的错误
IPE	0xE		指令权限等级错异常	指令执行中的错误
FPD	0xF		浮点指令未使能异常	系统调用和陷入
SXD	0x10		128 位向量扩展指令未使能异常	系统调用和陷入
ASXD	0x11		256 位向量扩展指令未使能异常	系统调用和陷入
FPE	0x12	0x0	基础浮点指令异常	需要软件修正的运算
VFPE		0x1	向量浮点指令异常	需要软件修正的运算
WPEF	0x13	0x0	取指监测点异常	系统调用和陷入
WPEM		0x1	load/store 操作监测点异常	系统调用和陷入
INT			中断	外部事件
TLBR			TLB 重填异常	地址转换异常
MERR			机器错误异常	数据完整性问题

中断

LoongArch 指令系统支持中断线的中断传递机制，共定义了 13 个中断，分别是：1 个核间中断 (IPI) ，1 个定时器中断 (TI) ，1 个性能监测计数溢出中断 (PMI) ，8 个外部硬中断 (HWI0~HWI7) ，2 个软中断 (SWI0~SWI1) 。其中所有中断线上的中断信号都采用电平中断，

且都是高电平有效。当有中断发生时，这种高电平有效中断方式输入给处理器的中断线上将维持高电平状态直至中断被处理器响应处理。无论中断源来自处理器核外部还是内部，是硬件还是软件置位，这些中断信号都被不间断地采样并记录到 CSR.ESTAT 中 IS 域的对应比特位上。这些中断均为可屏蔽中断，除了 CSR.CRMD 中的全局中断使能位 IE 外，每个中断各自还有其局部中断使能控制位，在 CSR.ECFG 的 LIE 域中。当 CSR.ESTAT 中 IS 域的某位为 1 且对应的局部中断使能和全局中断使能均有效时，处理器就将响应该中断，并进入中断处理程序入口处开始执行。

在支持多个中断源输入的指令系统中，需要规范在多个中断同时触发的情况下，处理器是否区别不同来源的中断的优先级。当采用非向量中断模式的时候，处理器通常不区别中断优先级，此时若需要对中断进行优先级处理，可以通过软件方式予以实现，其通常的实现方案是：

1. 软件随时维护一个中断优先级 (IPL)，每个中断源都被赋予特定的优先级。
2. 正常状态下，CPU 运行在最低优先级，此时任何中断都可触发。
3. 当处于最高中断优先级时，任何中断都被禁止。
4. 更高优先级的中断发生时，可以抢占低优先级的中断处理过程。

当采用向量中断模式的时候，处理器通常不可避免地需要依照一套既定的优先级规则来从多个已生效的中断源中选择一个，跳转到其对应的处理程序入口处。LoongArch 指令系统实现的是向量中断，采用固定优先级仲裁机制，具体规则是硬件中断号越大优先级越高，即 IPI 的优先级最高，TI 次之，…，SWI0 的优先级最低。

部分控制状态寄存器

处理上述提到的 PRMD 和 CRMD 两个寄存器外，实验中还会涉及到其它很多寄存器，下面只会显示本节可能会使用到的寄存器

ECONFIGURATION 寄存器 (ECFG)

该寄存器用于控制例外和中断的入口计算方式以及各中断的局部使能位。

表 7-6 例外配置寄存器定义

位	名字	读写	描述
12:0	LIE	RW	局部中断使能位，高有效。这些局部中断使能位与 CSR.ESTAT 中 IS 域记录的 13 个中断源一一对应，每一位控制一个中断源。
15:13	0	R0	保留域。读返回 0，且软件不允许改变其值。
18:16	VS	RW	配置例外和中断入口的间距。当 VS=0 时，所有例外和中断的入口地址是同一个。当 VS!=0 时，各例外和中断之间的入口地址间距是 2^{VS} 条指令。 因为 TLB 重填例外和机器错误例外其独立的入口基址，所以二者的例外入口不受 VS 域的影响。
31:19	0	R0	保留域。读返回 0，且软件不允许改变其值。

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。

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。

应用程序

应用程序的实现与 rCore 中是相同的，差异主要体现在编译和系统调用方式上，与内核编译代码一样，我们也要使用 build.rs 在编译前完成替换链接脚本和关闭标准库的工作，并且链接脚本的起始地址也被设置 0x00200000，此阶段内核代码不会超过此范围。

loongarch 寄存器

名称	别名	用途	在调用中是否保留
\$r0	\$zero	常数 0	(常数)
\$r1	\$ra	返回地址	否
\$r2	\$tp	线程指针	(不可分配)
\$r3	\$sp	栈指针	是
\$r4 - \$r5	\$a0 - \$a1	传参寄存器、返回值寄存器	否
\$r6 - \$r11	\$a2 - \$a7	传参寄存器	否
\$r12 - \$r20	\$t0 - \$t8	临时寄存器	否
\$r21		保留	(不可分配)
\$r22	\$fp / \$s9	栈帧指针 / 静态寄存器	是
\$r23 - \$r31	\$s0 - \$s8	静态寄存器	是

Table 2. 浮点寄存器使用约定

名称	别名	用途	在调用中是否保留
\$f0 - \$f1	\$fa0 - \$fa1	传参寄存器、返回值寄存器	否
\$f2 - \$f7	\$fa2 - \$fa7	传参寄存器	否
\$f8 - \$f23	\$ft0 - \$ft15	临时寄存器	否
\$f24 - \$f31	\$fs0 - \$fs7	静态寄存器	是

临时寄存器也被称为调用者保存寄存器。静态寄存器也被称为被调用者保存寄存器。

应用程序二进制接口

ABI 定义了应用程序二进制代码中数据结构和函数模块的格式及其访问方式，它使得不同的二进制模块之间的交互成为可能。硬件上并不强制这些内容，因此自成体系的软件可以不遵循部分或者全部 ABI 约定。但通常来说，应用程序至少会依赖操作系统以及系统函数库，因而必须遵循相关约定。

ABI 包括但不限于如下内容：

- 处理器基础数据类型的大小、布局和对齐要求等
- 寄存器使用约定。它约定通用寄存器的使用方法、别名等
- 函数调用约定。它约定参数如何传递给被调用的函数、结果如何返回、函数栈帧如何组织等
- 目标文件和可执行文件格式
- 程序装载和动态链接相关信息
- 系统调用和标准库接口定义
- 开发环境和执行环境等相关约定

loongArch 的函数调用规范包括了整型调用规范，浮点调用规范，而整型或浮点又会包含复合类型，在实验中，进行系统调用时我们传递都是整数，没有更复杂的类型，因此只需要关注整数部分。

基本整型调用规范提供了 8 个参数寄存器 \$a0-\$a7 用于参数传递，前两个参数寄存器 \$a0\$ 和 \$a1\$ 也用于返回值。若一个标量宽度至多 XLEN 位（对于 LP32 ABI，XLEN=32，对于 LPX32/LP64，XLEN=64），则它在单个参数寄存器中传递，若没有可用的寄存器，则在栈上传递。若一个标量宽度超过 XLEN 位，不超过 2*XLEN 位，则可以在一对参数寄存器中传递，低 XLEN 位在小编号寄存器中，高 XLEN 位在大编号寄存器中；若没有可用的参数寄存器，则在栈上传递标量；若只有一个寄存器可用，则低 XLEN 位在寄存器中传递，高 XLEN 位在栈上传递。若一个标量宽度大于 2*XLEN 位，则通过引用传递，并在参数列表中用地址替换。用栈传递的标量会对齐到类型对齐 (Type Alignment) 和 XLEN 中的较大者，但不会超过栈对齐要求。当整型参数传入寄存器或栈时，小于 XLEN 位的整型标量根据其类型的符号扩展至 32 位，然后符号扩展为 XLEN 位。以上的说明规定了 loongArch 体系结构下传递整数类型的方式，当前我们的系统工作在 64 位模式下，并且传递的参数不会超过 8 个，因此可以直接使用 8 个寄存器传递参数。

系统调用的汇编代码实现如下：

```
.section .text
.globl do_syscall
.align 4
do_syscall:
    #syscall(id: usize, args0: usize, args1: usize, args2: usize, )
    move $t0,$a0
    move $t1,$a1
    move $t2,$a2
    move $t3,$a3
    move $a7, $t0
    move $a0, $t1
    move $a1, $t2
    move $a2, $t3
    syscall 0
    jr $ra
```

至于为什么使用汇编文件编写而不是像 rCore 中一样使用内联汇编，原因在 debug 文档下有说明。这里不再赘述。在将各个参数放入寄存器后使用 `syscall` 指令来进行系统调用，这会触发一个系统调用异常，从而进入内核态。

特权级切换

通常异常和中断的处理对用户程序来说是透明的，相关软硬件需要保证处理前后原来执行中的代码看到的 CPU 状态保持一致。这意味着开始异常和中断处理程序之前需要保存所有可能被破坏的、原上下文可见的 CPU 状态，并在处理完返回原执行流之前恢复。需要保存的上下文包括异常处理代码的执行可能改变的寄存器、发生异常的地址、处理器状态寄存器、中断屏蔽位等现场信息以及特定异常的相关信息（如触发存储访问异常的地址）。异常和中断的处理代码通常在内核态执行，如果它们触发前处理器处于用户态，硬件会自动切换到内核态。这种情况下通常栈指针也会被重新设置为指向内核态代码所使用的栈，以便隔离不同特权等级代码的运行信息。

在本节中，当 CPU 在用户态特权级（loongarch 的 PLV3 模式）运行应用程序，执行到 Trap，切换到内核态特权级（loongarch 的 PLV0 模式），批处理操作系统的对应代码响应 Trap，并执行系统调用服务，处理完毕后，从内核态返回到用户态应用程序继续执行后续指令。

除了上篇文章提到的几个寄存器外，还需要使用到的寄存器包括：

BADV

该寄存器用于触发地址错误相关例外时，记录出错的虚地址。此类例外包括：

- 取指地址错例外 (ADEF)，此时记录的是该指令的 PC。
- load/store 操作地址错例外 (ADEM)
- 地址对齐错例外 (ALE)
- 边界约束检查错例外 (BCE)
- load 操作页无效例外 (PIL)
- store 操作页无效例外 (PIS)
- 取指操作页无效例外 (PIF)
- 页修改例外 (PME)
- 页不可读例外 (PNR)
- 页不可执行例外 (PNX)
- 页特权等级不合规例外 (PPI)

表 7-10 出错虚地址寄存器定义

位	名字	读写	描述
GRLEN-1:0	VAddr	RW	当触发地址错误相关例外时，硬件将出错的虚地址记录于此。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，那么记录的虚地址的高 32 位强制置为 0。

EENTRY

7.4.10 例外入口地址 (EENTRY)

该寄存器用于配置普通例外和中断的入口地址。

表 7-12 例外入口页号寄存器定义

位	名字	读写	描述
11:0	0	R	只读恒为 0, 写被忽略。
GRLEN-1:12	VPN	RW	普通例外和中断入口地址所在页的页号。

SAVE

7.4.16 数据保存 (SAVE)

数据保存控制状态寄存器用于给系统软件暂存数据。每个数据保存寄存器可以存放一个通用寄存器的数据。

数据保存寄存器最少实现 1 个, 最多实现 16 个。具体实现的个数软件可以从 CSR.PRCFG1.SAVENum 中获知。从 SAVE0 开始, 各个 SAVE 寄存器的地址依次为 0x30、0x31、……、0x30+SAVENum-1。

所有数据保存控制状态寄存器的格式均相同, 如表 7-18 所示。

表 7-18 数据保存寄存器定义

位	名字	读写	描述
GRLEN-1:0	Data	RW	仅供软件读写的数据。除执行 CSR 指令外, 硬件不会修改该域的内容。

TICLR

7.6.5 定时中断清除 (TICLR)

软件通过对该寄存器位 0 写 1 来清除定时器置起的定时中断信号。

表 7-48 定时中断清除寄存器定义

位	名字	读写	描述
0	CLR	W1	当对该 bit 写值 1 时, 将清除时钟中断标记。该寄存器读出结果总为 0。
31:1	0	R0	保留域。读返回 0, 且软件不允许改变其值。

上面的EENTRY是除了TLB重填例外和机器错误的入口地址, TLB 重填例外的入口来自于 CSR.TLBRETRY。机器错误例外的入口来自于 CSR.MERRETRY。例外的入口地址采用“入口页号 | 页内偏移”的计算方式。这里“|”是按位或运算, 所有普通例外入口的入口页号相同, 自于 CSR.EENTRY, 入口的偏移由中断偏移的模式和例外号 (ecode) 共同决定, 其值等于

$$2^{CSR.ECFG.VS+2} \times (code + 64)$$

当 CSR.ECFG.VS=0 时，所有普通例外的入口相同，此时需要软件通过 CSR.ESTAT 中的 Ecode、IS 域的信息来判断具体的例外类型。当 CSR.ECFG.VS !=0 时，不同的中断源具有不同的例外入口，软件无需通过访问 CSR.ESTAT 来确认例外类型。由于例外入口是基址“按位或”上偏移值，当 CSR.ECFG.VS !=0 时，软件在分配例外入口基址时需要确保所有可能的偏移值都不会超出入口基址低位所对应的边界对齐空间。在本实验中统一采用同一个入口地址。

当异常发生时，主要会经历下面的阶段：

1. 异常处理准备,当异常发生时，CPU 在转而执行异常处理前，硬件需要进行一系列准备工作。首先，需要记录被异常打断的指令的地址（记为 EPTR），TLB 重填异常发生时，这一信息将被记录在CSR.TLBRBERA 中；机器错误异常发生时，这一信息将被记录在 CSR.MERRERA 中，普通异常在CSR.ERA中。其次，调整 CPU 的权限等级（通常调整至最高特权等级）并关闭中断响应。在 LoongArch 指令系统中，当异常发生时，硬件会将 CSR.PLV 置 0 以进入最高特权等级，并将 CSR.CRMD 的 IE 域置 0 以屏蔽所有中断输入。再次，硬件保存异常发生现场的部分信息。在 LoongArch 指令系统中，异常发生时会将 CSR.CRMD 中的 PLV 和 IE 域的旧值分别记录到 CSR.PRMD 的 PPLV 和 PIE 域中，供后续异常返回时使用。最后，记录异常的相关信息。异常处理程序将利用这些信息完成或加速异常的处理。最常见的如记录异常编号以用于确定异常来源。在 LoongArch 指令系统中，这一信息将被记录在 CSR.ESTAT 的 Ecode 和 EsubCode 域，前者存放异常的一级编号，后者存放异常的二级编号。除此以外，有些情况下还会将引发异常的指令的机器码记录在 CSR.BADI 中，或是将造成异常的访存虚地址记录在 CSR.BADV 中。
2. 确定异常来源。不同类型的异常需要各自对应的异常处理。两种处理方式在介绍ENTRY已经说明。
3. 保存执行状态。在操作系统进行异常处理前，软件要先保存被打断的程序状态，通常至少需要将通用寄存器和程序状态字寄存器的值保存到栈中。也就是下文的trap上下文
4. 处理异常。跳转到对应异常处理程序进行异常处理。也就是下文的trap_handler函数
5. 恢复执行状态并返回。在异常处理返回前，软件需要先将前面第 3 个步骤中保存的执行状态从栈中恢复出来，在最后执行异常返回指令。之所以要采用专用的异常返回指令，是因为该指令需要原子地完成恢复权限等级、恢复中断使能状态、跳转至异常返回目标等多个操作。在 LoongArch 中，异常返回的指令是 ERTN，该指令会将 CSR.PRMD 的 PPLV 和 PIE 域分别回填至 CSR.CRMD 的 PLV 和 IE 域，从而使得 CPU 的权限等级和全局中断响应状态恢复到异常发生时的状态，同时该指令还会将 CSR.ERA 中的值作为目标地址跳转过去。

Trap上下文

rCore 中关于Trap上下文的定义如下：

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct TrapContext {
    pub x: [usize; 32], //通用寄存器，第4个寄存器是sp
    pub sstatus: usize, //控制状态寄存器
    pub sepc: usize, //异常处理返回地址
}
```

本实验中未做特殊的修改，但是其中sstatus保存的是PRMD的值，而sepc保存的是era的值。trap上下文的保存和恢复如下：

```
.section k_eentry
.globl __alltraps
.globl __restore
.align 4
__alltraps:
    # csrrw sp, sscratch, sp
    #需要交换 sp 与 0x502 寄存器的值
    csrwr $sp, 0x502

    addi.d $sp, $sp, -272
    #保存通用寄存器
    st.d $r0, $sp, 0
    st.d $r1, $sp, 8
    st.d $r2, $sp, 16
    # 这里不需要保存 sp的值
    #st.d $r3, $sp, 24 #sp
    st.d $r4, $sp, 32
    st.d $r5, $sp, 40
    st.d $r6, $sp, 48
    st.d $r7, $sp, 56
    st.d $r8, $sp, 64
    st.d $r9, $sp, 72
    st.d $r10, $sp, 80
    st.d $r11, $sp, 88
    st.d $r12, $sp, 96
    st.d $r13, $sp, 104
    st.d $r14, $sp, 112
    st.d $r15, $sp, 120
    st.d $r16, $sp, 128
    st.d $r17, $sp, 136
    st.d $r18, $sp, 144
    st.d $r19, $sp, 152
    st.d $r20, $sp, 160
    #st.d $r21, $sp, 168
    st.d $r22, $sp, 176
    st.d $r23, $sp, 184
    st.d $r24, $sp, 192
    st.d $r25, $sp, 200
    st.d $r26, $sp, 208
    st.d $r27, $sp, 216
    st.d $r28, $sp, 224
    st.d $r29, $sp, 232
    st.d $r30, $sp, 240
    st.d $r31, $sp, 248

    #读取crmd
    csrrd $t0, 0x1
    csrrd $t1, 0x6 #返回地址
    st.d $t0, $sp, 256
    st.d $t1, $sp, 264

    csrrd $t2, 0x502 #读出用户栈指针

    st.d $t2, $sp, 24
    # set input argument of trap_handler(cx: &mut TrapContext)

    move $a0, $sp
```

```

bl trap_handler

__restore:
    move $sp, $a0
    ld.d $t1, $sp, 264
    ld.d $t0, $sp, 256 #PRMD
    ld.d $t2, $sp, 24  #用户栈指针

    csrwr $t1, 0x6      #返回地址
    csrwr $t2, 0x502    #将用户栈指针放到DSAVE中
    csrwr $t0, 0x1      #恢复CRMD的值

    # 恢复通用寄存器
    ld.d $r0, $sp, 0
    ld.d $r1, $sp, 8
    ld.d $r2, $sp, 16
    # 这里不需要恢复 sp, 此时这个内存位置是用户栈指针
    # ld.d $r3, $sp, 24
    ld.d $r4, $sp, 32
    ld.d $r5, $sp, 40
    ld.d $r6, $sp, 48
    ld.d $r7, $sp, 56
    ld.d $r8, $sp, 64
    ld.d $r9, $sp, 72
    ld.d $r10, $sp, 80
    ld.d $r11, $sp, 88
    ld.d $r12, $sp, 96
    ld.d $r13, $sp, 104
    ld.d $r14, $sp, 112
    ld.d $r15, $sp, 120
    ld.d $r16, $sp, 128
    ld.d $r17, $sp, 136
    ld.d $r18, $sp, 144
    ld.d $r19, $sp, 152
    ld.d $r20, $sp, 160
    ld.d $r21, $sp, 168
    ld.d $r22, $sp, 176
    ld.d $r23, $sp, 184
    ld.d $r24, $sp, 192
    ld.d $r25, $sp, 200
    ld.d $r26, $sp, 208
    ld.d $r27, $sp, 216
    ld.d $r28, $sp, 224
    ld.d $r29, $sp, 232
    ld.d $r30, $sp, 240
    ld.d $r31, $sp, 248
    #r0不用恢复
    # release TrapContext on kernel stack
    addi.d $sp, $sp, 272

    csrwr $sp, 0x502
    #将用户栈指针与内核栈指针交换
    ertn

```

整体上来说与 rCore 中的类似，但这里使用的sscratch 寄存器被替换为DSAVE寄存器（0x502），这是一个调试数据保存寄存器，由于在qemu中检查到没有实现上述提到SAVE寄存

器，因此只能暂时使用此寄存器。

相对而言，初始化例外入口比 rCore 中要复杂，具体实现如下：

```
pub fn init() {
    extern "C" {
        fn __alltraps();
    }
    let mut ticlr = Ticlr::read();
    ticlr.clear();
    Ecfg::read().set_vs(0);
    Ecfg::read().set_local_interrupt(11, false);
    Crmd::read().set_interrupt_enable(false); //关闭全局中断
    Eentry::read().set_eentry(__alltraps as usize); // 设置中断入口
}
```

首先通过TICLR寄存器写入来消除时钟中断，再设置ECFG.vs = 0,使得所有例外的入口地址指向同一个，然后修改ECFG的中断配置，将已经开启的时钟中断关闭，修改CRMD关闭全局中断，这阶段不需要开启中断。最后设置中断入口。由于中断入口需要4k对齐，因此在链接脚本中需要将其设置对齐。

trap_handle的实现如下：

```

#[no_mangle]
pub fn trap_handler(cx: &mut TrapContext) -> &mut TrapContext {
    let prmd = Prmd::read().get_val(); //记录例外前的信息
    let estat = Estat::read();
    if (prmd & PRMD_PPLV) != 0 {
        // 非特权级0的例外
        // INFO!("kerneltrap: not from privilege0");
    }
    let crmd = Crmd::read();
    if crmd.get_interrupt_enable() {
        // 全局中断会在中断处理程序被关掉
        INFO!("kerneltrap: global interrupt enable");
    }
    match estat.cause() {
        Trap::Syscall => {
            //系统调用
            cx.sepc += 4;
            cx.x[4] = syscall(cx.x[11], [cx.x[4], cx.x[5], cx.x[6]]) as usize;
        }
        Trap::LoadPageFault | Trap::StorePageFault | Trap::FetchPageFault => {
            //页面异常
            println!("[kernel] PageFault in application, core dumped.");
            run_next_app();
        }
        Trap::InstructionNotExist => {
            //指令不存在
            println!("[kernel] IllegalInstruction in application, core
dumped.");
            run_next_app();
        }
        Trap::InstructionPrivilegeIllegal => {
            //指令权限不足
            println!("[kernel] InstructionPrivilegeIllegal in application, core
dumped.");
            run_next_app();
        }
        _ => {
            let mut record = 0;
            for i in 0..13{
                if estat.get_val().get_bit(i){
                    record = i;
                }
            }
            panic!(
                "Unsupported trap {:?}", interrupt = {}!",
                estat.get_val().get_bits(16..=21), record
            );
        }
    }
    cx
}

```

trap_handler的处理主要是读取estat,得到发生异常的原因进行相关处理。

为了进入用户态，需要在任务的trap上下文中保存PRMD的值，通过将其PPLV域设置为用户态，在trap上下文恢复完成后使用ertn指令即可将PPLV的值恢复到CRMD.PLV中，此时处理器就运行

在用户态下了。

第三章

第三章主要是完成多道程序加载和分时共享，需要了解的知识包括

1. 任务切换与trap上下文切换的区别
2. 被调用者保存寄存器
3. 时钟中断相关寄存器
4. 计时器

任务切换

在这一章的第一小节，我们需要完成一个多道程序系统，相对于第二章的批处理系统，多道程序可以更有效地利用内存空间，由于各个应用程序位于不同的物理地址上，也就免去了刷新Cache带来的开销。而为了可以达到同时加载多个应用程序的功能，需要在生成应用程序时指定各个程序应该放置的内存位置，在实验中使用了一个python脚本完成。从编译的角度也可以看到，这样的多道程序对编写程序的人的限制很大，因为大多数人并不事先知道计算机上哪一部分的内存可用，因此如果想要放入计算机运行，还需要提前修改编译的条件。而要解决这个问题，需要在第四章完成地址空间的抽象和硬件地址转换结构的支持下可以。

任务切换是第二章提及的 Trap 控制流切换之外的另一种异常控制流，都是描述两条控制流之间的切换，如果将它和 Trap 切换进行比较，会有如下异同：

- 与 Trap 切换不同，它不涉及特权级切换；
- 与 Trap 切换不同，它的一部分是由编译器帮忙完成的；
- 与 Trap 切换相同，它对应用是透明的。

大致来看其与一般的函数调用是非常相似的，差别仅仅在于任务切换时会发生栈的替换。根据 loongarch 的寄存器规定，32 个寄存器中被调用者保存寄存器为 s0-s9, 还需要保存 ra 与 sp 的值，因此任务上下文的定义如下：

```
/// 任务上下文
/// 对于一般的函数，编译器会在函数的起始位置自动生成代码保存 被调用者保存寄存器
/// _switch 函数不会被编译器特殊处理，因此我们需要手动保存这些寄存器
/// 而其它寄存器不保存时因为属于调用者保存的寄存器是由编译器在高级语言编写
/// 的调用函数中自动生成的代码来完成保存的；还有一些寄存器属于临时寄存器，
/// 不需要保存和恢复。
#[derive(Copy, Clone, Debug)]
#[repr(C)]
pub struct TaskContext {
    //ra: 此寄存器存储的是函数返回时跳转的地址
    //在调用函数返回指令时，Pc 指针会跳转到 ra 里面的地址
    ra: usize,
    sp: usize,
    s: [usize; 10], //loongArch 下需要保存 10 个 s 寄存器
}
```

同时还需要修改 trap.S 的实现，具体而言是删除掉 `__restore` 开头部分的

```
move $sp, $a0
```

`__restore` 在正常的运行过程中可能被调用的情况包含两种：

1. 从正常的 trap 上下文保存 `_alltraps` 结束返回后继续执行
2. 应用程序在第一次调用 `_switch` 时会跳转到 `__restore` 执行，由于这个时候 `_switch` 已经发生了内核栈的切换，并且此时切换到的任务就是即将要执行的任务，加载的的 sp 值为初始化的应用程序内核栈保存 trap 上下文的位置，因此在执行到 `__restore` 时，sp 已经是正确的值，就无需再次进行从 a0 加载。

```
for i in 0..num_app {
    tasks[i].task_cx_ptr = TaskContext::goto_restore(init_app(cx(i)));
    tasks[i].task_status = TaskStatus::Ready;
} // 初始化任务上下文
```

```
/// 用于初始化相关的设置
pub fn init_app(cx(app: usize) -> usize {
    // 返回任务的上下文
    let t = KERNEL_STACK[app].push_context(
        // 首先压入trap上下文，再压入task上下文
        TrapContext::app_init_context(get_base_address(app),
USER_STACK[app].get_sp())
    );
    t
}
```

```
pub fn app_init_context(entry: usize, sp: usize) -> Self {
    let sstatus = Crmd::read();
    let sstatus = sstatus.get_val();
    Prmd::read().set_pplv(3);
    // 设置进入用户态
    // 应该在汇编代码中切换到用户态
    let mut cx = Self {
        x: [0; 32],
        sstatus,
        sepc: entry,
    };
    cx.set_sp(sp);
    cx
}
```

```
pub fn goto_restore(kstack_ptr: usize) -> Self {
    extern "C" {
        fn __restore();
    }
    Self {
        ra: __restore as usize,
        sp: kstack_ptr, // 存放了trap上下文后的栈地址，内核栈地址
        s: [0; 10],
    }
}
```

时钟

在多道程序和协作式调度的基础上，应用程序可以各自获得处理器的使用权了，但这仍然需要应用程序的编写者主动地去让出处理器，如果某个应用程序不主动让出，那么其它的任务将永远不会获得使用权。为了使得操作系统对应用程序的管理更加公平合理，需要完成

- 分时多任务：操作系统管理每个应用程序，以时间片为单位来分时占用处理器运行应用。
- 时间片轮转调度：操作系统在一个程序用完其时间片后，就抢占当前程序并调用下一个程序执行，周而复始，形成对应用程序在任务级别上的时间片轮转调度。

在前文中已经大致介绍了loongarch上的中断，各中断源发来的中断信号被处理器采样至 CSR.ESTAT.IS 域中，这些信息与软件配置在 CSR.ECFG.LIE 域中的局部中断使能信息按位与，得到一个 13 位中断向量 int_vec。当 CSR.CRMD.IE=1 且 int_vec 不为全0 时，处理器认为有需要响应的中断，于是从执行的指令流中挑选出一条指令，将其标记上一种特殊的例外——中断例外。随后处理器硬件的处理过程与普通例外的处理过程一样了。

loongarch平台上的时钟中断与risc-v的不相同，与时钟中断相关的寄存器如下：

TCFG

该寄存器是软件配置定时器的接口。定时器的有效位数由实现决定，因此该寄存器中 TimeVal 域的位宽也将随之变化。

表 7-45 定时器配置寄存器定义

位	名字	读写	描述
0	En	RW	定时器使能位。仅当该位为 1 时，定时器才会进行倒计时自减，并在减为 0 值时置起定时中断信号。
1	Periodic	RW	定时器循环模式控制位。若该位为 1，定时器在倒计时自减至 0 时，在置起定时中断信号的同时，还会自动定时器重新装载成 InitVal 域中配置的初始值，然后再下一个时钟周期继续自减。若该位为 0，定时器在倒计时自减至 0 时，将停止计数直至软件再次配置该定时器。
n-1:2	InitVal	RW	定时器倒计时自减计数的初始值。要求该初始值必须是 4 的整倍数。硬件将自动在该域数值的最低位补上两比特 0 后再使用。
GRLEN-1:n	0	R	只读恒为 0，写被忽略。

TVAL

7.6.3 定时器数值 (TVAL)

软件可通过读取该寄存器来获知定时器当前的计数值。定时器的有效位数由实现决定，因此该寄存器中 TimeVal 域的位宽也将随之变化。

表 7-46 定时器剩余寄存器定义

位	名字	读写	描述
n-1:0	TimeVal	R	当前定时器的计数值。
GRLEN-1:n	0	R	只读恒为 0，写被忽略。

为了打开时钟中断，不仅需要开启全局中断和时钟中断对应的局部使能中断，还需要配置 CSR.TCFG 中的 En 位。时钟的中断频率由 TCFG.Initval 位决定，在 risc-v 中每次发生时钟中断，都需要配置 mtimecmp 寄存器的值，在 loongarch 上，只需要开启 TCFG.Periodic 位就可以使得在发生时钟中断后不用在手动更新下一次发生中断的时间。

开启时钟中断的代码如下：

```
pub fn enable_timer_interrupt() {
    Ticlr::read().clear(); // 清除时钟专断
    Tcfg::read()
        .set_enable(true) // 开启计时
        .set_loop(true) // 开启循环
        .set_tval(0x100000usize) // 设置中断间隔时间
        .flush(); // 写入计时器的配置
    Ecfg::read().set_local_interrupt(11, true); // 开启局部使能中断
    Crmd::read().set_interrupt_enable(true); // 开启全局中断
}
```

时钟中断对应的处理代码如下：

```
fn timer_handler() {
    let mut ticlr = Ticlr::read();
    ticlr.set_val(ticlr.get_val() | CSR_TICLR_CLR); // 清除时钟中断
    suspend_current_run_next();
}
```

龙芯指令系统定义了一个恒定频率计时器，其主体是一个 64 位的计数器，称为 Stable Counter。Stable Counter 在复位后置为 0，随后每个计数时钟周期自增 1，当计数至全 1 时自动绕回至 0 继续自增。同时每个计时器都有一个软件可配置的全局唯一编号，称为 Counter ID。恒定频率计时器的特点在于其计时频率在复位后保持不变，无论处理器核的时钟频率如何变化。获取计时器的实现如下：

```
pub struct Time {}

impl Time {
    pub fn read() -> usize {
        let mut counter:usize;
        unsafe {
            asm!(
                "rdtime.d {},{}",
                out(reg)counter,
                out(reg)_,
            );
        }
        counter
    }
}
```

第四章

第四章是在loongarch平台上遇到的与大量硬件相关的第一次尝试，这一章中，不仅需要了解loongarch上大量的寄存器以及其功能，并且需要知道risc-v和其在地址空间管理上的差别，比如映射地址空间，以及手动管理TLB等。而且由于开启了页表的缘故，debug的过程也可能比较艰难，因此需要细细品读相关的细节。

- 了解寄存器设计，为大部分寄存器实现相应的接口，屏蔽掉访问的复杂性
- 介绍内存分配的一点点知识并给出相关知识链接
- 介绍loongarch的存储管理
- 详细介绍loongarch的页表机制
- 完成多级页表的软件实现，需要修改很多代码
- 完成TLB重填和页修改异常的处理
- 介绍如何配置多级页表

寄存器设计

在 risc-v 或者 x86 平台，rust 均有对应的库支持，里面包含各种寄存器操作或者 IO 操作的抽象，而对 loongarch 平台的支持的库少之又少，在前面的实验中虽然对 loongarch 的部分寄存器也进行了抽象，但使用起来仍然比较不方便，因此在开始这一章的实验前，需要对 loongarch 平台上的相关寄存器添加支持，类似于建立一个 crate，方便后面的代码对寄存器进行操作。

loongarch 下的控制状态寄存器包含如下：

地址	名称	
0x0	当前模式信息	CRMD
0x1	例外前模式信息	PRMD
0x2	扩展部件使能	EUEN
0x3	杂项控制	MISC
0x4	例外配置	ECFG
0x5	例外状态	ESTAT
0x6	例外返回地址	ERA
0x7	出错虚地址	BADV
0x8	出错指令	BADI
0xc	例外入口地址	EENTRY
0x10	TLB 索引	TLBIDX
0x11	TLB 表项高位	TLBEHI
0x12	TLB 表项低位 0	TLBELO0
0x13	TLB 表项低位 1	TLBELO1
0x18	地址空间标识符	ASID
0x19	低半地址空间全局目录基址	PGDL
0x1A	高半地址空间全局目录基址	PGDH

0x1B	全局目录基址	PGD
0x1C	页表遍历控制低半部分	PWCL
0x1D	页表遍历控制高半部分	PWCH
0x1E	STLB 页大小	STLBPS
0x1F	缩减虚地址配置	RVACFG
0x20	处理器编号	CPUID
0x21	特权资源配置信息 1	PRCFG1

0x22	特权资源配置信息 2	PRCFG2
0x23	特权资源配置信息 3	PRCFG3
0x30+n (0≤n≤15)	数据保存	SAVEn
0x40	定时器编号	TID
0x41	定时器配置	TCFG
0x42	定时器值	TVAL
0x43	计时器补偿	CNTC
0x44	定时中断清除	TICLR
0x60	LLBit 控制	LLBCTL
0x80	实现相关控制 1	IMPCTL1
0x81	实现相关控制 2	IMPCTL2
0x88	TLB 重填例外入口地址	TLBREENTRY
0x89	TLB 重填例外出错虚地址	TLBRBADV
0x8A	TLB 重填例外返回地址	TLBRERA
0x8B	TLB 重填例外数据保存	TLBRSAVE
0x8C	TLB 重填例外表项低位 0	TLBRELO0
0x8D	TLB 重填例外表项低位 1	TLBRELO1
0x8E	TLB 重填例外表项高位	TLBREHI

0x8F	TLB 重填例外前模式信息	TLBRPRMD
0x90	机器错误控制	MERRCTL
0x91	机器错误信息 1	MERRINFO1
0x92	机器错误信息 2	MERRINFO2
0x93	机器错误例外入口地址	MERRENTRY
0x94	机器错误例外返回地址	MERRERA
0x95	机器错误例外数据保存	MERRSAVE
0x98	高速缓存标签	CTAG
0x180+n (0≤n≤3)	直接映射配置窗口 n	DMWn
0x200+2n (0≤n≤31)	性能监测配置 n	PMCFGn
0x201+2n (0≤n≤31)	性能监测计数器 n	PMCNTn
0x300	load/store 监视点整体控制	MWPC
0x301	load/store 监视点整体状态	MWPS
0x310+8n (0≤n≤7)	load/store 监视点 n 配置 1	MWPnCFG1
0x311+8n (0≤n≤7)	load/store 监视点 n 配置 2	MWPnCFG2

0x312+8n (0≤n≤7)	load/store 监视点 n 配置 3	MWPnCFG3
0x313+8n (0≤n≤7)	load/store 监视点 n 配置 4	MWPnCFG4
0x380	取指监视点整体控制	FWPC
0x381	取指监视点整体状态	FWPS
0x390+8n (0≤n≤7)	取指监视点 n 配置 1	FWPnCFG1
0x391+8n (0≤n≤7)	取指监视点 n 配置 2	FWPnCFG2
0x392+8n (0≤n≤7)	取指监视点 n 配置 3	FWPnCFG3
0x393+8n (0≤n≤7)	取指监视点 n 配置 4	FWPnCFG4
0x500	调试寄存器	DBG
0x501	调试例外返回地址	DERA
0x502	调试数据保存	DSAVE

这些寄存器各个位上的含义各不相同，并且同一个寄存器各个位的读取属性也不一样，具体的读写属性有四种：

1. RW——软件可读、可写。除在定义中明确指出的会导致处理器执行结果不确定的非法值，软件可以写入任意值。通常情况下，软件对这些域进行先写后读的操作，读出的应该是写入的值。但是，当所访问的域可以被硬件更新时，或者执行读、写操作的两条指令之间有中断发生，则有可能出现读出值与写入值不一致的情况。
2. R——软件只读。软件写这些域不会更新其内容，且不产生其它任何副作用
3. RO——软件读取这些域永远返回 0。但是同时软件必须保证，要么通过设置 CSR 写屏蔽位避免更新这些域，要么在更新这些域时必须要写入 0 值。这一要求是为了确保软件向后兼容。对于硬件实现来说，标记这种属性的域将禁止软件写入。
4. W1——软件写 1 有效。软件对这些域写 0 不会将其清 0，且不产生其它任何副作用。同时，定义为该属性的域的读出值没有任何软件意义，软件应该无视这些读出值。

所有控制状态寄存器的位宽，或者固定为 32 位，或者与所实现的是 LA32 还是 LA64 相关。对于第一种类别的寄存器，其在 LA64 架构下被 CSR 指令访问时，读返回的是符号扩展至 64 位后的值，写的时候高 32 位的值自动被硬件忽略。对于第二种类型，定义将明确指出 LA32 和 LA64 架构下的差异。当软件使用 CSR 指令访问的 CSR 对象是架构规范中未定义的，或者是架构规范中定义的可实现项但是具体硬件未实现的，此时读动作返回的可以是任意值，但写动作不应改变软件可见的处理器状态。

在前面的章节中我们已经看到了部分寄存器的使用，通常，对所有可用的寄存器都会实现

Register trait，其定义如下：

```
pub trait Register {  
    fn read() -> Self;  
    fn write(&mut self);  
}
```

因此，如果想要写一个寄存器必须将其先读出，虽然这可能带来一定的性能损失，因为可能有人会选择直接设置寄存器的值，但经过思考，大部分控制状态寄存器的值是不会被修改的，只有某些位的值需要我们修改，因此可以在读出其值的情况下再去设置某个位，这样就避免了手动查看寄存器各个位的默认值然后再设置整个值带来的复杂性，因此这里选择了这种读出写入的模式。

比如 CRMD 寄存器各个位的定义如下：

表 7-2 当前模式信息寄存器定义

位	名字	读写	描述
1:0	PLV	RW	<p>当前特权等级。其合法的取值范围为 0~3。其中 0 表示最高特权等级，3 表示最低特权等级。</p> <p>当触发例外时，硬件将该域的值置为 0，以确保陷入后处于最高特权等级。</p> <p>当执行 ERTN 指令从例外处理程序返回时， 如果 CSR. ERRCTL. IsMERR=1，则硬件将 CSR. ERRCTL 的 PPLV 域的值恢复到这里； 否则，如果 CSR. TLBRERA. IsTLBR=1，则硬件将 CSR. TLBRPRMD 的 PPLV 域的值恢复到这里； 否则，硬件将 CSR. PRMD 的 PPLV 域的值恢复到这里。</p>
2	IE	RW	<p>当前全局中断使能，高有效。</p> <p>当触发例外时，硬件将该域的值置为 0，以确保陷入后屏蔽中断。例外处理程序决定重新开启中断响应时，需显式地将该位置 1。</p> <p>当执行 ERTN 指令从例外处理程序返回时， 如果 CSR. ERRCTL. IsMERR=1，则硬件将 CSR. ERRCTL 的 PIE 域的值恢复到这里； 否则，如果 CSR. TLBRERA. IsTLBR=1，则硬件将 CSR. TLBRPRMD 的 PIE 域的值恢复到这里； 否则，硬件将 CSR. PRMD 的 PIE 域的值恢复到这里。</p>
3	DA	RW	<p>直接地址翻译模式的使能，高有效。</p> <p>当触发 TLB 重填例外或是机器错误例外时，硬件将该域置为 1。</p> <p>当执行 ERTN 指令从例外处理程序返回时，</p>

			如果 CSR. ERRCTL. IsMERR=1，则硬件将 CSR. ERRCTL 的 PDA 域的值恢复到这里；否则，如果 CSR. TLBRERA. IsTLBR=1，则硬件将该域置为 0。 DA 位和 PG 位的合法组合情况为 0、1 或 1、0，当软件配置成其它组合情况时结果不确定。
4	PG	RW	映射地址翻译模式的使能，高有效。 当触发 TLB 重填例外或是机器错误例外时，硬件将该域置为 0。 当执行 ERTN 指令从例外处理程序返回时， 如果 CSR. ERRCTL. IsMERR=1，则硬件将 CSR. ERRCTL 的 PPG 域的值恢复到这里；否则，如果 CSR. TLBRERA. IsTLBR=1，则硬件将该域置为 1。 PG 位和 DA 位的合法组合情况为 0、1 或 1、0，当软件配置成其它组合情况时结果不确定。
6:5	DATF	RW	直接地址翻译模式时，取指操作的存储访问类型。 当触发机器错误例外时，硬件将该域置为 0。 当执行 ERTN 指令从例外处理程序返回，且 CSR. ERRCTL. IsMERR=1，则硬件将 CSR. ERRCTL 的 PDATF 域的值恢复到这里。 在采用软件处理 TLB 重填的情况下，当软件将 PG 置为 1 时，需同时将 DATF 域置为 0b01，即一致可缓存类型。
8:7	DATM	RW	直接地址翻译模式时，load 和 store 操作的存储访问类型。 当触发机器错误例外时，硬件将该域置为 0。 当执行 ERTN 指令从例外处理程序返回，且 CSR. ERRCTL. IsMERR=1，则硬件将 CSR. ERRCTL 的 PDATM 域的值恢复到这里。 在采用软件处理 TLB 重填的情况下，当软件将 PG 置为 1 时，需同时将 DATM 域置为 0b01，即一致可缓存类型。
9	WE	RW	指令和数据监视点的使能位，高电平有效。 当触发例外时，硬件将该域的值置为 0。 当执行 ERTN 指令从例外处理程序返回时， 如果 CSR. ERRCTL. IsMERR=1，则硬件将 CSR. ERRCTL 的 PWE 域的值恢复到这里；否则，如果 CSR. TLBRERA. IsTLBR=1，则硬件将 CSR. TLBRPRMD 的 PWE 域的值恢复到这里； 否则，硬件将 CSR. PRMD 的 PWE 域的值恢复到这里。
31:10	0	RO	保留域。读返回 0，且软件不允许改变其值。

那么针对它的寄存器实现如下：

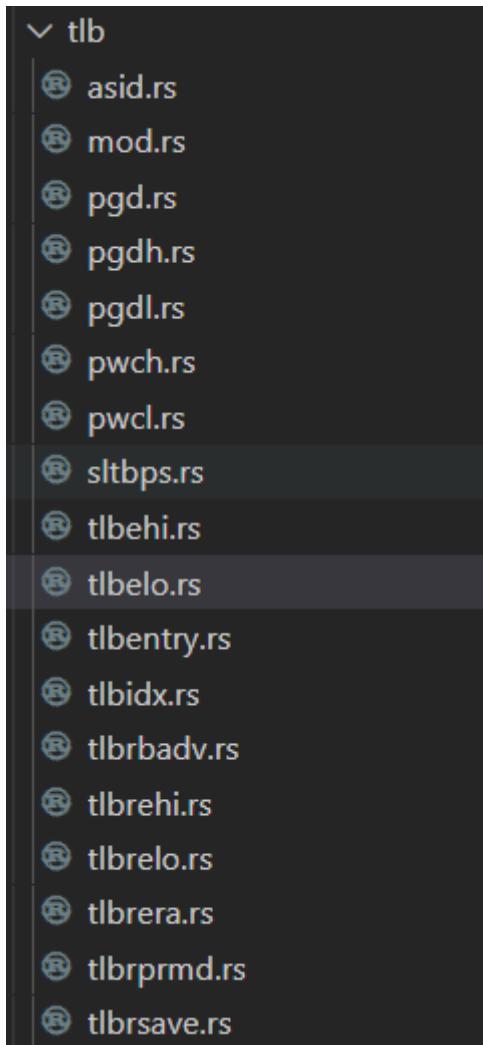
```
✓ Crmd
  ↘ bits usize
✓ {} impl Register for Crmd
  ↗ read fn() -> Self
  ↗ write fn(&mut self)
✓ {} impl Crmd
  ↗ get_val fn(&self) -> usize
  ↗ set_val fn(&mut self, val: usize) -> &mut Self
  ↗ get_plv fn(&self) -> usize
  ↗ set_plv fn(&mut self, mode: CpuMode) -> &mut Self
  ↗ set_ie fn(&mut self, enable: bool) -> &mut Self
  ↗ get_ie fn(&self) -> bool
  ↗ get_da fn(&self) -> bool
  ↗ set_da fn(&mut self, da: bool) -> &mut Self
  ↗ get_pg fn(&self) -> bool
  ↗ set_pg fn(&mut self, pg: bool) -> &mut Self
  ↗ get_datf fn(&self) -> usize
  ↗ set_datf fn(&mut self, datf: usize) -> &mut Self
  ↗ get_datm fn(&self) -> usize
  ↗ set_datm fn(&mut self, datm: usize) -> &mut Self
```

实现中会针对寄存器的各个有效域和对应的读写属性提供相应的函数，用户在使用时可以直接对想要读写的域进行操作，而且对大多数寄存器而言都会提供 `get_val` 和 `set_val` 两个函数，这两个函数通常为直接想要设置寄存器整个值的人使用，在C语言编写的操作系统通常会使用或操作来设置整个寄存器的值。

其它的寄存器与上面的实现是类似的，这里就不再给出各个寄存器的详细实现，可以查看源代码以获取更多详细。实验中实现了图标给出的大部分寄存器，部分未使用到的寄存器也可以按照上述方法较为容易实现。这里给出实验中完成的寄存器：

register

- ② badi.rs
- ② badv.rs
- ② cpuid.rs
- ② crmd.rs
- ② csr.rs
- ② dmwn.rs
- ② ecfg.rs
- ② eentry.rs
- ② era.rs
- ② estat.rs
- ② mod.rs
- ② prcfg1.rs
- ② prcfg2.rs
- ② prcfg3.rs
- ② prmd.rs
- ② rvacfg.rs
- ② saven.rs
- ② tcfg.rs
- ② ticlr.rs
- ② time.rs



loonarch定义了一个CPUCFG指令，用于软件在执行过程中动态识别所运行的处理器中实现了龙芯架构中的哪些功能特性。这些指令系统功能特性的实现情况记录在一系列配置信息字中，CPUCFG 指令执行一次可以读取一个配置信息字。

配置信息字中包含一系列配置位（域），其记录形式为 CPUCFG.<配置字号>.<配置信息助记名称>[位下标]，其中单比特配置位的位下标记为 bitXX，表示配置字的第 XX 位；多比特的配置域的位下标记为 bitXX:YY，表示配置字的第 XX 位到第 YY 位的连续(XX-YY+1)位。例如，1 号配置字中的第 0 位用以表示是否实现 LA32 架构，将这个配置信息记录为 CPUCFG.1.LA32[bit0]，其中 1 表示配置信息字的字号是 1 号，LA32 表示这个配置信息域所起的助记名称叫做 LA32，bit0 表示 LA32 这个域位于配置字的第 0 位。1 号配置字中第 11 位到第 4 位的记录所支持物理地址位数的 PALEN 域则记为 CPUCFG.1.PALEN[bit11:4]

配置字包含的信息很多，可以查看提供的文档了解各个配置字含义，这里给出第 1，2 个配置字各位的含义，在本章节中，只实现了几个暂时需要使用的配置字。

表 2-2 CPUCFG 访问配置信息列表

字号	位下标	助记名称	含义
0	31:0	PRID	处理器标识
1	1:0	ARCH	2' b00 表示实现 LA32 精简架构; 2' b01 表示实现 LA32 架构; 2' b10 表示实现 LA64 架构。2' b11 保留。
	2	PGMMU	为 1 表示 MMU 支持页映射模式
	3	IOCSR	为 1 表示支持 IOCSR 指令
	11:4	PALEN	所支持的物理地址位数 PALEN 的值减 1
	19:12	VALEN	所支持的虚拟地址位数 VALEN 的值减 1
	20	UAL	为 1 表示支持非对齐访存
	21	RI	为 1 表示支持“读禁止”页属性
	22	EP	为 1 表示支持“执行保护”页属性
	23	RPLV	为 1 表示支持 RPLV 页属性
	24	HP	为 1 表示支持 huge page 页属性
	25	IOCSR_BRD	为 1 表示 IOCSR 访问空间的 0 地址处记录了处理器产品信息的字符串。即 “Loongson3A5000 @ 2.5GHz”这样的信息。
	26	MSG_INT	为 1 表示外部中断采用消息中断方式，否则为电平中断线方式

对配置字的抽象如下：

```
pub struct CPUCFG {
    bits: usize,
}

impl CPUCFG {
    // 读取index对应字的内容
    pub fn read(index: usize) -> Self {
        let mut bits;
        unsafe {
            asm!("cpucfg {},{}", out(reg) bits, in(reg) index);
        }
        Self { bits }
    }
    pub fn get_bit(&self, index: usize) -> bool {
        self.bits.get_bit(index)
    }
    pub fn get_bits(&self, start: usize, end: usize) -> usize {
        self.bits.get_bits(start..=end)
    }
}
```

当我们想获取支持的虚拟地址位数，需要提供的实现如下：

```
pub fn get_valen() -> usize {
    let cfg = CPUCFG::read(1);
    cfg.get_bits(12, 19) + 1
}
```

在实验中，在完成中断和异常初始化后会打印当前机器的相关信息，从这个函数中我们可以看到各个寄存器和cpu配置字的使用方法

```

// 打印硬件的相关信息
pub fn print_machine_info() {
    info!("PALEN: {}", get_palen()); //支持的物理地址范围
    info!("VALEN: {}", get_valen()); //支持的虚拟地址范围
    info!("Support MMU-Page :{}", get_mmu_support_page());
    info!("Support Read-only :{}", get_support_read_forbid());
    info!(
        "Support Execution-Protect :{}",
        get_support_execution_protection()
    ); //是否支持执行保护页属性
    info!("Support RPLV: {}", get_support_rplv()); //是否支持吃rplv页属性
    info!("Support RVA: {}", get_support_rva()); //是否支持虚拟地址缩减
    info!("Support RVAMAX :{}", get_support_rva_len()); //支持的虚拟地址缩减的长度
    info!("Support Page-Size: {:#b}", Prcfg2::read().get_val()); //支持的页大小,
    match Prcfg3::read().get_tlb_type() {
        0 => {
            info!("No TLB");
        }
        1 => {
            info!("Have MTLB");
        }
        2 => {
            info!("Have STLB + MTLB");
        }
        _ => {
            info!("Unknown TLB");
        }
    }
    info!("MLTB Entry: {}", Prcfg3::read().get_mtlb_entries()); //MTLB的页数量
    info!("SLTB Ways :{}", Prcfg3::read().get_stlb_ways()); //STLB的路数量
    info!("SLTB Entry: {}", Prcfg3::read().get_sltb_sets()); //STLB每一路的项数
    info!("SLTB Page-size: {}", SltbPs::read().get_page_size()); //STLB的页大小
    info!("PTE-size: {}", Pwcl::read().get_pte_width()); //PTE的大小
    info!("TLB-RFill entry_point: {:#x}", TLBREEntry::read().get_val()); //TLB重填的入口地址
    info!("TLB-RFill page-size :{}", TlbREhi::read().get_page_size()); //TLB重填的页大小
    let pwcl = Pwcl::read();
    info!(
        "PTE-index-width: {},{}",
        pwcl.get_ptbase(),
        pwcl.get_ptwidth()
    ); //PTE的索引宽度
    info!(
        "PGD-index-width: {},{}",
        pwcl.get_dir1_base(),
        pwcl.get_dir1_width()
    ); //PGD的索引宽度
    let pwch = Pwch::read();
    info!(
        "PMD-index-width: {},{}",
        pwch.get_dir3_base(),
        pwch.get_dir3_width()
    ); //PTE的索引宽度
    let crmd = Crmd::read();
    info!("DA: {}", crmd.get_da()); //是否支持DA模式
    info!("PG :{}", crmd.get_pg()); //是否支持PG模式
}

```

```
    info!("dmwo: {:?}", Dmw0::read().get_value()); //映射窗口1
    info!("dmw1: {:?}", Dmw1::read().get_value()); //映射窗口2
    info!("PLV: {}", crmd.get_plv()); //当前的特权级
}
```

有了上述实现的支持，在后续的实现中就可以很方便地查看和配置loongarch机器上各种属性了。

内存分配算法

我们目前在内核中使用两种类型的变量：局部变量和static变量。局部变量存储在调用堆栈中，并且仅在周围函数返回之前有效。静态变量存储在固定的内存位置，并且在程序的整个生命周期中始终存在。

局部变量和静态变量结合在一起已经非常强大，并且支持大多数用例。但是，我们看到它们都有其局限性：

- 局部变量只存在于周围的函数或块的末尾。这是因为它们存在于调用堆栈上，并在周围的函数返回后被销毁。
- 静态变量始终存在于程序的整个运行时，因此在不再需要它们时无法回收和重用它们的内存。此外，它们的所有权语义不明确，并且可以从所有函数访问，因此当我们想要修改它们时，它们需要受到锁的保护。

局部变量和静态变量的另一个限制是它们具有固定的大小。因此，它们无法存储在添加更多元素时动态增长的集合。为了避免这些缺点，编程语言通常支持第三个内存区域来存储变量，称为**堆**。堆通过两个名为! `allocate` 和 `deallocate` 的函数在运行时支持动态内存分配。`allocate` 的工作方式如下：该函数返回一个指定大小的空闲内存块，可用于存储变量。然后这个变量一直存在，直到通过调用 `deallocate` 释放它。

在C语言中，堆的使用往往带来很多致命性的问题，包括内存泄漏，use-after-free, 双重释放等，而一些语言为了避免这种问题，通常采用垃圾回收的技术自动管理动态内存，但这也带来了很大的性能开销。rust在两者之间找到了一个平衡点，它使用一个称为所有权的概念，能够在编译时检查动态内存操作的正确性。因此，不需要垃圾收集来避免上述问题，这意味着没有性能开销。这种方法的另一个优点是程序员仍然可以对动态内存的使用进行细粒度控制，就像使用 C 或 C++ 一样。

Rust 标准库没有让程序员手动调用 `allocate` 和 `deallocate`，而是提供了隐式调用这些函数的抽象类型，这些类型在创建时会申请内存空间，失效时会自动释放内存，这些类型通常实现了 `Drop trait`。常见的抽象类型有 `Box`, `Arc`, `Rc`, `cell` 等。在这些抽象类型的基础上，可以使用多种数据集合，诸如 `Vec<T>`, `BTreeMap<K, V>`, `String` 等，有了这些类型和集合的支持，操作系统的内核将可以提供更多的功能。

要在操作系统内核中支持动态内存分配，则需要实现一系列必要的功能：初始化堆、分配/释放内存块的函数接口、连续内存分配算法。相对于 C 语言而言，Rust 语言在 `alloc crate` 中设定了一套简洁规范的接口，只要实现了这套接口，内核就可以很方便地支持动态内存分配了。

`alloc` 库需要我们提供给它一个 全局的动态内存分配器，它会利用该分配器来管理堆空间，从而使得与堆相关的智能指针或容器数据结构可以正常工作。具体而言，我们的动态内存分配器需要实现它提供的 `GlobalAlloc Trait`，这个 Trait 有两个必须实现的抽象接口：

```
// alloc::alloc::GlobalAlloc

pub unsafe fn alloc(&self, layout: Layout) -> *mut u8;
pub unsafe fn deallocate(&self, ptr: *mut u8, layout: Layout);
```

两个接口中都有一个 `alloc::alloc::Layout` 类型的参数，它指出了分配的需求，分为两部分，分别是所需空间的大小 `size`，以及返回地址的对齐要求 `align`。这个对齐要求必须是一个 2 的幂次，单位为字节数，限制返回的地址必须是 `align` 的倍数。

本实验中有两个可用的内存分配器，一个是来自rCore官方的 `buddy-system-allocator`，一个是我自己实现的分配器，与官方实现一样，也用到了类似伙伴分配器的思想，当然实现要简单的多，但由于实现中使用了大量的unsafe和裸指针，因此这里就暂且不介绍其实现了，感兴趣的同学可以查看源代码，相比rcore官方来说，可能更为容易理解。而且通过测试，两个分配器算法的效率大致是相同的。

在文末的链接中提供了C语言实现的多个分配器，感兴趣的读者可以阅读并移植到rust上。

链接

- [liballoc](#) Excellent allocator that was originally a part of the Spoon Operating System and designed to be plugged into hobby OS's.
- [dlmalloc](#) - Doug Lea's Memory Allocator. A good all purpose memory allocator that is widely used and ported.
- [TCMalloc](#) Thread-Caching Malloc. An experimental scalable allocator.
- [nedmalloc](#) A very fast and very scalable allocator. These two properties have made it somewhat popular in multi-threaded video games as an alternative to the default provided allocator.
- [ptmalloc](#) A widely used memory allocator included with glibc that scales reasonably while being space efficient.
- [jemalloc](#) A general purpose malloc(3) implementation that emphasizes fragmentation avoidance and scalable concurrency support, first used in FreeBSD
- [Hoard](#) is a drop-in replacement for malloc that can dramatically improve application performance, especially for multithreaded programs running on multiprocessors and multicore CPUs.
- [Bucket](#) is a simple drop-in replacement for malloc for beginners. Most importantly, it has detailed documentation of what happens under the hood. Not just source code comments.

存储管理

loongarch管理的内存物理地址空间范围是： $0 - 2^{PALEN-1}$ 。在 LA32 架构下，PALEN 理论上是一个不超过 36 的正整数，由实现决定其具体的值，通常建议为 32。在 LA64 架构下，PALEN 理论上是一个不超过 60 的正整数，由实现决定其具体的值。系统软件可以通过 CPUCFG 读取配置字 0x1 的 PALEN 域来确定 PALEN 的具体值。

loongarch架构中虚拟地址空间是线性平整的。对于 PLV0 级来说，LA32 架构下虚拟地址空间大小为 2^{32} 字节，LA64 架构下虚拟地址空间大小为 2^{64} 字节。不过对于 LA64 架构来说， 2^{64} 字节大小的虚拟地址空间并不都是合法的，可以认为存在一些虚拟地址的空洞。合法的虚拟地址空间与地址映射模式紧密相关。并且对于应用程序来说，在 LA32 架构下，应用软件能够访问的内存地址空间范围是： $0 - 2^{31} - 1$ ，在 LA64 架构下，应用软件能够访问的内存地址空间范围是： $0 - 2^{VALEN} - 1$ 。这里 VALEN 理论上是一个小于等于 64 的整数，由实现决定其具体的值，应用软件可以通过执行 CPUCFG 指令读取 0x1 号配置字的 VALEN 域来确定 VALEN 的具体值。

loongarch的MMU 支持两种虚实地址翻译模式：直接地址翻译模式和映射地址翻译模式

当 CSR.CRMD 的 DA=1 且 PG=0 时，处理器核的 MMU 处于直接地址翻译模式。在这种映射模式下，物理地址默认直接等于虚拟地址的[PALEN-1:0]位（不足补 0），除非具体实现中采用了其它优先级更高的虚实地址翻译规则。可以看到此时整个虚拟地址空间都是合法的。处理器复位结束后将进入直接地址翻译模式。当 CSR.CRMD 的 DA=0 且 PG=1 时，处理器核的 MMU 处于映射地址翻译模式。具体又分为直接映射地址翻译模式（简称“直接映射模式”）和页表映射地址翻译模式（简称“页表映射模式”）两种。翻译地址时将优先看其能否按照直接映射模式进行翻译，无法进行后再按照页表映射模式进行翻译。

loongarch架构下支持三种存储访问类型，分别是：一致可缓存（Coherent Cached，简称 CC）、强序非缓存（Strongly-ordered UnCached，简称 SUC）和弱序非缓存（Weakly-ordered UnCached，简称 WUC）。当处理器核 MMU 处于直接地址翻译模式时，所有取指的存储访问类型由 CSR.CRMD.DATF 决定，所有 load/store 操作的存储访问类型由 CSR.CRMD.DATM 域决定。当处理器核 MMU 处于映射地址翻译模式时，存储访问类型的确定分为两种情况。如果取指或 load/store 操作的地址落在某个直接映射配置窗口上，那么该取指或 load/store 操作的存储访问类型由配置该窗口的 CSR 寄存器中的 MAT 域决定。如果取指或 load/store 只能通过页表完成映射，那么其存储访问类型由页表项中的 MAT 域决定。无论在哪种情况下，存储访问类型控制值的定义是相同的，均是：

0——强序非缓存，1——一致可缓存，2——弱序非缓存，3——保留。

直接映射地址翻译模式

当处理器核的 MMU 处于映射地址模式时，还可以通过直接映射配置窗口机制完成虚实地址的直接映射。直接映射配置窗口共设置有四个，前两个窗口可同时用于取指和 load/store 操作，后两个窗口仅用于 load/store 操作。系统软件通过配置 CSR.DMW0~CSR.DMW3 寄存器来分别设置四个直接映射配置窗口。每个窗口除了地址范围信息外，还可以配置该窗口在哪些特权等级下可用，以及虚地址落在该窗口上的访存操作的存储访问类型。

这一组寄存器参与完成直接映射地址翻译模式。有关该地址翻译模式的内容请参看 5.2.1 节。

表 7-42 直接映射配置窗口寄存器定义 (LA64 架构)

位	名字	读写	描述
0	PLV0	RW	为 1 表示在特权等级 PLV0 下可以使用该窗口的配置进行直接映射地址翻译。
1	PLV1	RW	为 1 表示在特权等级 PLV1 下可以使用该窗口的配置进行直接映射地址翻译。
2	PLV2	RW	为 1 表示在特权等级 PLV2 下可以使用该窗口的配置进行直接映射地址翻译。
3	PLV3	RW	为 1 表示在特权等级 PLV3 下可以使用该窗口的配置进行直接映射地址翻译。
5:4	MAT	RW	虚地址落在该映射窗口下访存操作的存储访问类型。
59:6	0	R0	保留域。读返回 0，且软件不允许改变其值。
63:60	VSEG	RW	直接映射窗口的虚地址的[63:60]位。

在 LA64 架构下，每一个直接映射配置窗口可以配置一个 2^{PALEN} 字节固定大小的虚拟地址空间。当虚地址命中某个有效的直接映射配置窗口时，其物理地址直接等于虚地址的[PALEN-1:0]位。命中的判断方式是：虚地址最高 4 位 ([63:60]位) 与配置窗口寄存器中的 VSEG 域相等，且当前特权等级在该配置窗口中被允许。

举例来说，在 PALEN 等于 48 的情况下，通过将 DMW0 配置为 0x9000000000000011，那么在 PLV0 级下，0x9000000000000000 ~ 0x9000FFFFFFFFFFFF 这段原本在页映射模式下不合法的虚地址空间，将被映射到物理地址空间 0x0 ~ 0xFFFFFFFFFFFF 上，且存储访问类型是一致可缓存的。

页表映射存储管理

映射地址翻译模式下，除了落在直接映射配置窗口中的地址之外，其余所有合法地址都必须通过页表映射完成虚实地址转换。TLB 作为处理器中存放操作系统页表信息的一个临时缓存，用于加速映射地址翻译模式下的取指和 load/store 操作的虚实地址转换过程。

关于页表的内容在下一小节将着重介绍。

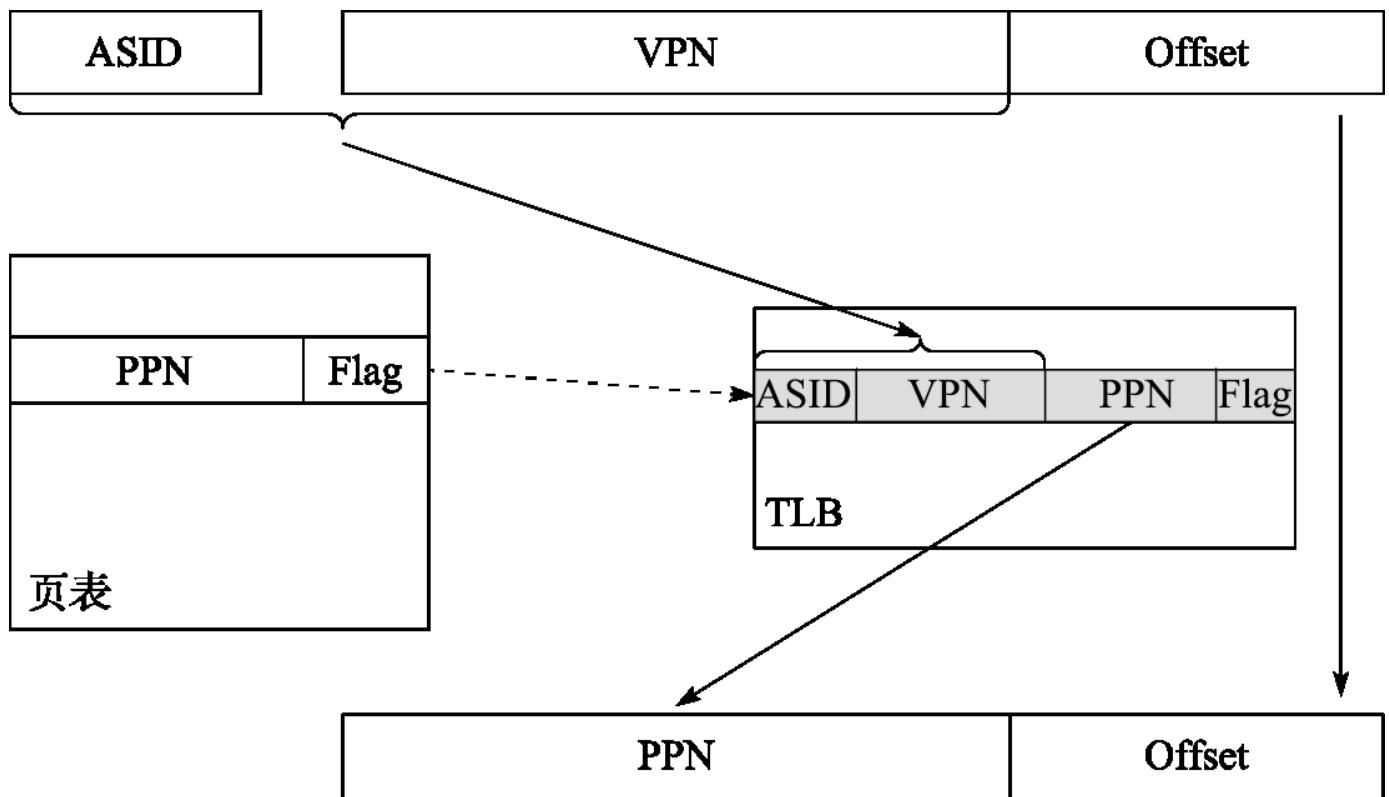
多级页表硬件机制

处理器的存储管理部件（Memory Management Unit，简称 MMU）支持虚实地址转换、多进程空间等功能，是通用处理器体现“通用性”的重要单元，也是处理器和操作系统交互最紧密的部分。存储管理构建虚拟的内存地址，并通过 MMU 进行虚拟地址到物理地址的转换。存储管理的作用和意义包括以下方面。

- 隐藏和保护：用户态程序只能访问受限内存区域的数据，其他区域只能由核心态程序访问。引入存储管理后，不同程序仿佛在使用独立的内存区域，互相之间不会影响。此外，分页的存储管理方法对每个页都有单独的写保护，核心态的操作系统可防止用户程序随意修改自己的代码段
- 为程序分配连续的内存空间：MMU 可以由分散的物理页构建连续的虚拟内存空间，以页为单元管理物理内存分配
- 扩展地址空间：在 32 位系统中，如果仅采用线性映射的虚实地址映射方式，则至多访问 4GB 物理内存空间，而通过 MMU 进行转换则可以访问更大的物理内存空间
- 节约物理内存：程序可以通过合理的映射来节约物理内存。当操作系统中有相同程序的多个副本在同时运行时，让这些副本使用相同的程序代码和只读数据是很直观的空间优化措施，而通过存储管理可以轻松完成这些。此外，在运行大型程序时，操作系统无须将该程序所需的所有内存都分配好，而是在确实需要使用特定页时再通过存储管理的相关异常处理来进行分配，这种方法不但节约了物理内存，还能提高程序初次加载的速度。

为了提高页表访问的速度，现代处理器中通常包含一个转换后援缓冲器（Translation Lookaside Buffer，简称 TLB）来实现快速的虚实地址转换。TLB 也称页表缓存或快表，借由局部性原理，存储当前处理器中最经常访问页的页表。一般 TLB 访问与 Cache 访问同时进行，而 TLB 也可以被视为页表的 Cache。TLB 中存储的内容包括虚拟地址、物理地址和保护位，可分别对应于 Cache 的 Tag、Data 和状态位。

包含 TLB 的地址转换过程如下图所示：



处理器用地址空间标识符 (Address Space Identifier, 简称 ASID) 和虚拟页号 (Virtual Page Number, 简称 VPN) 在 TLB 中进行查找匹配，若命中则读出其中的物理页号 (Physical Page Number, 简称 PPN) 和标志位 (Flag)。标志位用于判断该访问是否合法，一般包括是否可读、是否可写、是否可执行等，若非法则发出非法访问异常；物理页号用于和页内偏移 (Offset) 拼接组成物理地址。若未在 TLB 中命中，则需要将页表内容从内存中取出并填入 TLB 中，这一过程通常称为 TLB 重填 (TLB Refill)。TLB 重填可由硬件或软件进行，例如 X86、ARM 处理器采用硬件 TLB 重填，即由硬件完成页表遍历 (Page Table Walker)，将所需的页表项填入 TLB 中；而 MIPS、LoongArch 处理器默认采用软件 TLB 重填，即查找 TLB 发现不命中时，将触发 TLB 重填异常，由异常处理程序进行页表遍历并进行 TLB 填入。

页表映射模式存储管理的核心部件是 TLB。LoongArch 指令系统下 TLB 分为两个部分，一个是所有表项的页大小相同的单一页面大小 TLB (Singular-Page-Size TLB, 简称 STLB)，另一个是支持不同表项的页大小可以不同的多重页面大小 TLB (Multiple-Page-Size TLB, 简称 MTLB)。STLB 的页大小可通过 STLBPS 控制寄存器进行配置。

在虚实地址转换过程中，STLB 和 MTLB 同时查找。相应地，软件需保证不会出现 MTLB 和 STLB 同时命中的情况，否则处理器行为将不可知。MTLB 采用全相联查找表的组织形式，STLB 采用多路组相联的组织形式。对于 STLB，如果其有 2^{INDEX} 组，且配置的页大小为 2^{PS} 字节，那么硬件查询 STLB 的过程中，是将虚地址的 PS+index:PS 位作为索引来访问各路信息。

TLB 表项

STLB 和 MTLB 的表项格式基本一致，区别仅在于 MTLB 每个表项均包含页大小信息，而 STLB 因为是同一页大小所以 TLB 表项中不再需要重复存放页大小信息。

VPPN	PS	G	ASID	E
PPN0	RPLV0 PLV0 MAT0 NX0 NR0 D0 V0			
PPN1	RPLV1 PLV1 MAT1 NX1 NR1 D1 V1			

图 3.4: LoongArch64 指令系统中 TLB 表项结构

- 存在位(E), 1 比特。为 1 表示所在 TLB 表项非空，可以参与查找匹配。
- 地址空间标识(ASID)，10 比特。地址空间标识用于区分不同进程中的同样的虚地址，避免进程切换时清空整个 TLB 所带来的性能损失。操作系统为每个进程分配唯一的 ASID，TLB 在进行查找时除了比对地址信息一致外，还需要比对 ASID 信息
- 全局标志位(G), 1 比特。当该位为 1 时，查找时不进行 ASID 是否一致性的检查。当操作系统需要在所有进程间共享同一虚拟地址时，可以设置 TLB 页表项中的 G 位置为 1。
- 页大小(PS), 6 比特。仅在 MTLB 中出现。用于指定该页表项中存放的页大小。数值是页大小的 2 的幂指数。即对于 16KB 大小的页，PS=14
- 虚双页号(VPPN), (VALEN-PS-1)比特。在龙芯架构中，每一个页表项存放了相邻的一对奇偶相邻页表信息，所以 TLB 页表项中存放虚页号的是系统中虚页号/2 的内容，即虚页号的最低位不需要存放在 TLB 中。查找 TLB 时在根据被查找虚页号的最低位决定是选择奇数号页还是偶数号页的物理转换信息
- 有效位(V), 1 比特。为 1 表明该页表项是有效的且被访问过的
- 脏位(D), 1 比特。为 1 表示该页表项所对应的地址范围内已有脏数据。
- 不可读位(NR), 1 比特。为 1 表示该页表项所在地址空间上不允许执行 load 操作。该控制位仅定义在 LA64 架构下。
- 不可执行位(NX), 1 比特。为 1 表示该页表项所在地址空间上不允许执行取指操作。该控制位仅定义在 LA64 架构下
- 存储访问类型(MAT), 2 比特。控制落在该页表项所在地址空间上访存操作的存储访问类型
- 特权等级 (PLV) , 2 比特。该页表项对应的特权等级。当 RPLV=0 时，该页表项可以被任何特权等级不低于 PLV 的程序访问；当 RPLV=1 时，该页表项仅可以被特权等级等于 PLV 的程序访问
- 受限特权等级使能 (RPLV) , 1 比特。页表项是否仅被对应特权等级的程序访问的控制位。请参看上面 PLV 中的内容。该控制位仅定义在 LA64 架构下
- 物理页号(PPN), (PALEN-12)比特。当页大小大于 4KB 的时候，TLB 中所存放的 PPN 的[PS-1:12]位可以是任意值

用 TLB 进行虚实地址翻译时，首先要进行 TLB 查找，将待查虚地址 vaddr 和 CSR.ASID 中 ASID 域的值 asid 一起与 STLB 中每一路的指定索引位置项以及 MTLB 中的所有项逐项进行比对。如果 TLB 表项的 E 位为 1，且 vaddr 对应的虚双页号 vppn 与 TLB 表项的 VPPN 相等（该比较需要根据 TLB 表项对应的页大小，只比较地址中属于虚页号的部分），且 TLB 表项中的 G 位为 1 或者 asid 与 TLB 表项的 ASID 域的值相等，那么 TLB 查找命中该 TLB 表项。如果没有命中项，则触发 TLB 重填异常 (TLBR) 。

如果查找到一个命中项，那么根据命中项的页大小和待查虚地址确定 vaddr 具体落在双页中的哪一页，从奇偶两个页表项取出对应页表项作为命中页表项。如果命中页表项的 V 等于 0，说明该

页表项无效，将触发页无效异常，具体将根据访问类型触发对应的 load 操作页无效异常 (PIL)、store 操作页无效异常 (PIS) 或取指操作页无效异常 (PIF)。

如果命中页表项的 V 值等于 1，但是访问的权限等级不合规，将触发页权限等级不合规异常（PPI）。权限等级不合规体现为，该命中页表项的 RPLV 值等于 0 且 CSR.CRMD 中 PLV 域的值大于命中页表项中的 PLV 值，或是该命中页表项的 RPLV=1 且 CSR.CRMD 中 PLV 域的值不等于命中页表项中的 PLV 值。

如果上述检查都合规，还要进一步根据访问类型进行检查。如果是一个 load 操作，但是命中页表项中的 NR 值等于 1，将触发页不可读异常（PNR）；如果是一个 store 操作，但是命中页表项中的 D 值等于 0，将触发页修改异常（PME）；如果是一个取指操作，但是命中页表项中的 NX 值等于 1，将触发页不可执行异常（PNX）。

如果找到了命中项目经检查上述异常都没有触发，那么命中项中的 PPN 值和 MAT 值将被取出，前者用于和 vaddr 中提取的页内偏移拼合成物理地址 paddr，后者用于控制该访问操作的内存访问类型属性。

LoongArch 指令系统中用于访问和控制 TLB 的控制状态寄存器大致可以分为三类：第一类用于非 TLB 重填异常处理场景下的 TLB 访问和控制，包括 TLBIDX、TLBEHI、TLBELO0、TLBELO1、ASID 和 BADV；第二类用于 TLB 重填异常处理场景，包括此场景下 TLB 访问控制专用的 TLBREHI、TLBRELO0、TLBRELO1 和 TLBRBADV 以及此场景下保存上下文专用的 TLBRPRMD、TLBRERA 和 TLBRSAVE；第三类用于控制页表遍历过程，包括 PGDL、PGDH、PGD、PWCL 和 PWCH。

TLBIDX		N E	0	PS	0	Index							
TLBEHI	Sign_Ext	VPPN					0						
TLBELO0	RP N N LV X R	0	PPN					0	G MAT PLV D V				
TLBELOI	RP N N LV X R	0	PPN					0	G MAT PLV D V				
ASID				0	ASIDBITS	0	ASID						
BADV	VAddr												
TLBREHI	Sign_Ext	VPPN					0	PS					
TLBRELO0	RP N N LV X R	0	PPN					0	G MAT PLV D V				
TLBRELOI	RP N N LV X R	0	PPN					0	G MAT PLV D V				
TLBRBADV	VAddr												
TLBRPRMD				0			PW E	0	P I E	PPLV			
TLBRERA	PC[63:2]												
TLBRSAVE	Data												
PGDL	Base					0							
PGDH	Base					0							
PGD	Base					0							
PWCL			PTE width	Dir2_width	Dir2_base	Dir1_width	Dir1_base	PTwidth	PTbase				
PWCH			0	Dir4_width	Dir4_base	Dir3_width	Dir3_base						

上述寄存器中，第二类专用于 TLB 重填异常处理场景 (CSR.TLBRERA 的 IsTLBR 域值等于 1) 的控制寄存器，其设计目的是确保在非 TLB 重填异常处理程序执行过程中嵌套发生 TLB 重填异常处理。后，原有异常处理程序的上下文不被破坏。例如，当发生 TLB 重填异常时，其异常处理返回地址将填入 CSR.TLBRERA 而非 CSR.ERA，这样被嵌套的异常处理程序返回时所用的返回目标就不会被破坏。因硬件上只维护了这一套保存上下文专用的寄存器，所以需要确保在 TLB 重填异常处

理过程中不再触发 TLB 重填异常，为此，处理器因 TLB 重填异常触发而陷入异常处理后，硬件会自动将虚实地址翻译模式调整为直接地址翻译模式，从而确保 TLB 重填异常处理程序第一条指令的取指和访存一定不会触发 TLB 重填异常，与此同时，软件设计人员也要保证后续 TLB 重填异常处理返回前的所有指令的执行不会触发 TLB 重填异常。

当触发 TLB 重填异常时，除了更新 CSR.CRMD 外，CSR.CRMD 中 PLV、IE 域的旧值将被记录到 CSR.TLBRPRMD 的相关域中，异常返回地址也将被记录到 CSR.TLBRERA 的 PC 域中，处理器还会将引发该异常的访存虚地址填入 CSR.TLBRBAV 的 VAddr 域并从该虚地址中提取虚双页号填入 CSR.TLBREHI 的 VPPN 域。当触发非 TLB 重填异常的其他 TLB 类异常时，除了像普通异常发生时一样更新 CRMD、PRMD 和 ERA 这些控制状态寄存器的相关域外，处理器还会将引发该异常的访存虚地址填入 CSR.BADV 的 VAddr 域并从该虚地址中提取虚双页号填入 CSR.TLBEHI 的 VPPN 域。

为了对 TLB 进行维护，除了上面提到的 TLB 相关控制状态寄存器外，LoongArch 指令系统中还定义了一系列 TLB 访问和控制指令，主要包括 TLBRD、TLBWR、TLBFILL、TLBSRCH 和 INVTLB。

TLBSRCH 为 TLB 查找指令，其使用 CSR.ASID 中 ASID 域和 CSR.TLBEHI 中 VPPN 域的信息（当处于 TLB 重填异常处理场景时，这些值来自 CSR.ASID 和 CSR.TLBREHI）去查询 TLB。如果有命中项，那么将命中项的索引值写入 CSR.TLBIDX 的 Index 域，同时将其 NE 位置为 0；如果没有命中项，那么将该寄存器的 NE 位置 1。

TLBRD 是读 TLB 的指令，其用 CSR.TLBIDX 中 Index 域的值作为索引读出指定 TLB 表项中的值并将其写入 CSR.TLBEHI、CSR.TLBEL00、CSR.TLBEL01 以及 CSR.TLBIDX 的对应域中。

TLBWR 是写 TLB 的指令，其用 CSR.TLBIDX 中 Index 域的值作为索引将 CSR.TLBEHI、CSR.TLBEL00、CSR.TLBEL01 以及 CSR.TLBIDX 相关域的值（当处于 TLB 重填异常处理场景时，这些值来自 CSR.TLBREHI、CSR.TLBRELO0 和 CSR.TLBRELO1）写到对应的 TLB 表项中。

在实验中，上述三个指令用于 TLB 页修改异常的处理中。

TLBFILL 是填入 TLB 的指令，其将 CSR.TLBEHI、CSR.TLBEL00、CSR.TLBEL01 以及 CSR.TLBIDX 相关域的值（当处于 TLB 重填异常处理场景时，这些值来自 CSR.TLBREHI、CSR.TLBRELO0 和 CSR.TLBRELO1）填入 TLB 中的一个随机位置。该位置的具体确定过程是，首先根据被填入页表项的页大小来决定是写入 STLB 还是 MTLB。当被填入的页表项的页大小与 STLB 所配置的页大小（由 CSR.STLBPS 中 PS 域的值决定）相等时将被填入 STLB，否则将被填入 MTLB。页表项被填入 STLB 的哪一路，或者被填入 MTLB 的哪一项，是由硬件随机选择的。

INVTLB 指令用于无效 TLB 中符合条件的表项，即从通用寄存器 rj 和 rk 得到用于比较的 ASID 和虚地址信息，依照指令 op 立即数指示的无效规则，对 TLB 中的表项逐一进行判定，符合条件的 TLB 表项将被无效掉。

多级页表结构

loongarch 处理器与 risc-v 处理器不同之处在于 risc-v 是根据 satp 寄存器的 Mode 域决定分页机制，而 loongarch 上在获取到其虚拟地址位宽后可以设置不同的页大小从而得到不同的多级页表结构，

如果其有效虚地址位宽为 48 位，那么当操作系统采用 16KB 页大小时，其页表为三级结构，如下图所示：

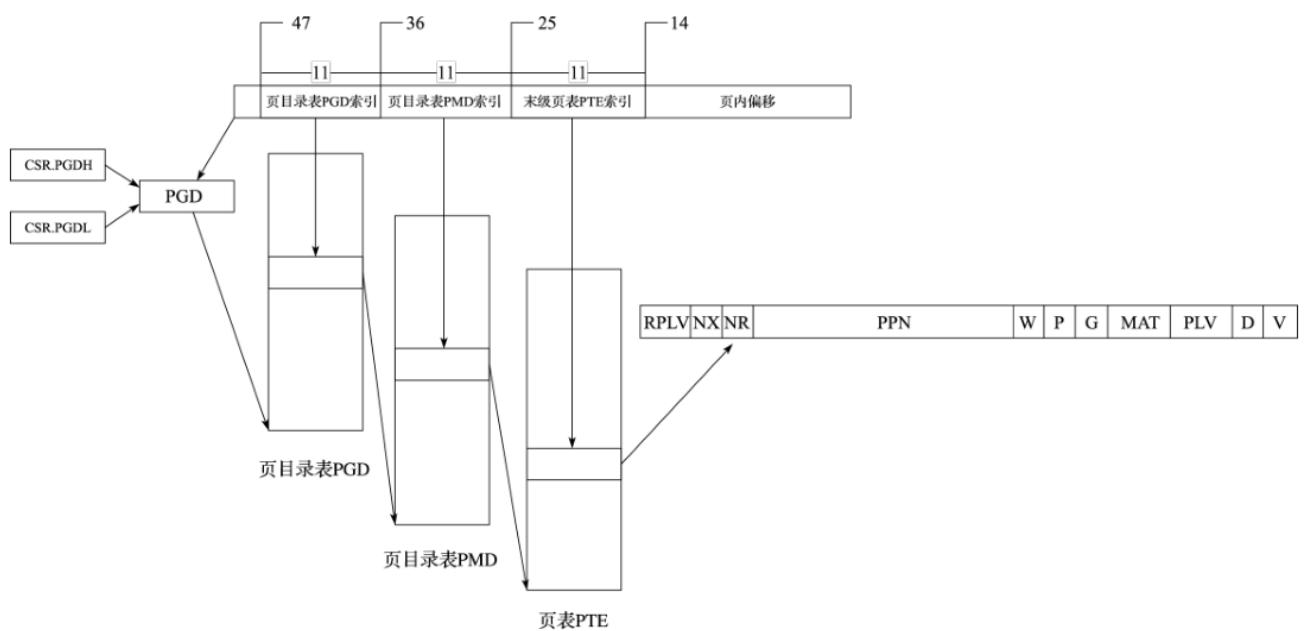


图 3.6: Linux/LoongArch 三级页表结构

33 位的虚双页号 (VPPN) 分为三个部分：最高 11 位作为一级页表（页目录表 PGD）索引，一级页表中每一项保存一个二级页表（页目录表 PMD）的起始地址；中间 11 位作为二级页表索引，二级页表中每一项保存一个三级页表（末级页表 PTE）的起始地址；最低 11 位作为三级页表索引。每个三级页表包含 2048 个页表项，每个页表项管理一个物理页，大小为 8 字节，包括 RPLV、NX、NR、PPN、W、P、G、MAT、PLV、D、V 的信息。“P”和“W”两个域分别代表物理页是否存在，以及该页是否可写。这些信息虽然不填入 TLB 表项中，但用于页表遍历的处理过程。每个进程的 PGD 表基地址放在进程上下文中，内核进程进行切换时把 PGD 表的基地址写到 CSR.PGDH 的 Base 域中，用户进程进行切换时把 PGD 表的基地址写到 CSR.PGDL 的 Base 域中。在实验中我们只会使用 PGDL，上述说明用于完善的操作系统中，因为完善的操作系统内核会被放置于内存的高位区域中，应用程序位于低地址区域中，但本实验中内核与应用程序都位于同一段物理内存区间。

页表项的定义总共包含两种，上图是其中一种：

基本页页表项格式：

63	62	61	PALEN-1	12	8	7	6	5	4	3	2	1	0
RPLV	NX	NR		PA[PALEN-1:12]			W	P	G	MAT	PLV	D	V

大页页表项格式：

63	62	61	PALEN-1	24	12	8	7	6	5	4	3	2	1	0
RPLV	NX	NR		PA[PALEN-1:24]		G		W	P	H	MAT	PLV	D	V

这里我们不关注大页页表项格式。

因此除了三级页表外，如果设置页大小不同，那么得到的多级页表页不相同，但总体而言，多级页表的格式如下：

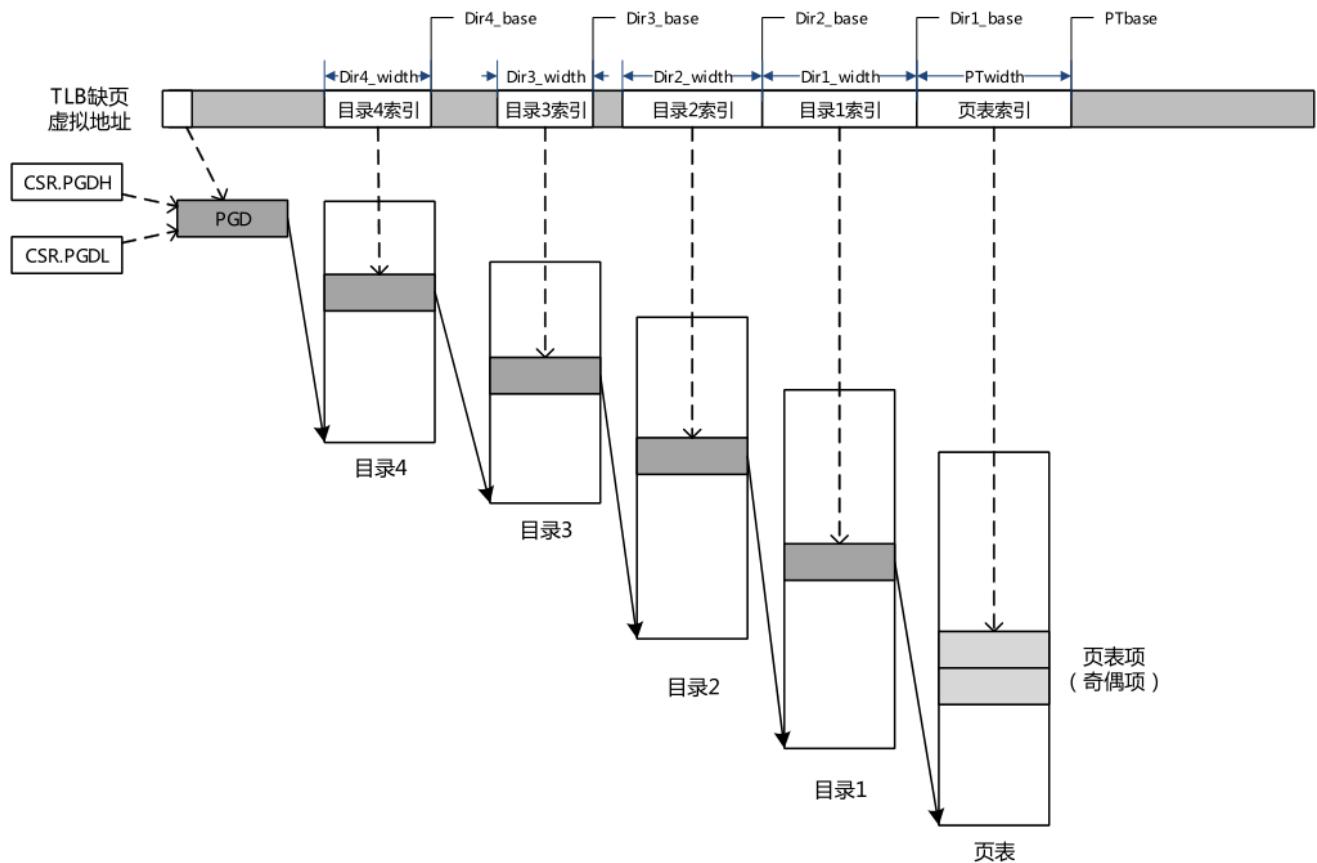


图 5-2 页表遍历过程所支持的多级页表结构

本实验会采用页大小为16kb的页，使用三级页表完成。

多级页表实现

通过CPUCFG指令获取系统配置后，我们得到了loongarch机器支持的虚拟地址区间位宽和物理地址区间均为48位，在将页大小规定为16kb后，将会构成三级页表机制。

在地址相关的数据结构抽象与类型定义中，只需要简单修改config文件中的页大小和位宽即可。

```
pub const PAGE_SIZE: usize = 0x4000; //16kB
pub const PAGE_SIZE_BITS: usize = 14; //16kB
```

在页表项的实现中，由于两个平台差异较大，因此许多结构需要重新定义，对于页表项中的标志位，重新定义如下：

```
bitflags! {
    pub struct PTEFlags: usize {
        const V = 1 << 0;
        const D = 1 << 1;
        const PLVL = 1 << 2;
        const PLVH = 1 << 3;
        const MATL = 1 << 4;
        const MATH = 1 << 5;
        const G = 1 << 6;
        const P = 1 << 7;
        const W = 1 << 8;
        const NR = 1 << 61;
        const NX = 1 << 62;
        const RPLV = 1 << 63;
    }
}
```

上述所有位都不会被硬件自动修改，在发生TLB相关的异常时，我们可能需要手动查找到页表项并进行修改，主要涉及的是D位。同时PTEFlags提供了默认的实现：

```
impl PTEFlags {
    fn default() -> Self {
        PTEFlags::V | PTEFlags::MATL | PTEFlags::P | PTEFlags::W
    }
}
```

在后续构建页表项时，需要获取默认实现再根据读写权限修改其它位，这也就是说，我们规定了这些页表项都是有效的，并且存储类型为一致可缓存，同时这些虚拟页号对应物理页均存在(P)，该页可写(W)。上文提到的其中P和W位只存在于页表项中，不会放入TLB中，这两个位主要用于cow机制中，但本实验不会涉及cow，因此这里将其规定为特定值。

对页表项相关的函数也做了修改：

```
impl PageTableEntry {
    pub fn new(ppn: PhysPageNum, flags: PTEFlags) -> Self {
        let mut bits = 0usize;
        bits.set_bits(14..PALEN, ppn.0); //采用16kb大小的页
        bits = bits | flags.bits;
        PageTableEntry { bits }
    }
    // 空页表项
    pub fn empty() -> Self {
        PageTableEntry { bits: 0 }
    }
    // 返回物理页号---页表项
    pub fn ppn(&self) -> PhysPageNum {
        self.bits.get_bits(14..PALEN).into()
    }
    // 返回物理页号---页目录项
    // 在一级和二级页目录表中目录项存放的是只有下一级的基地址
    pub fn directory_ppn(&self) -> PhysPageNum {
        (self.bits >> PAGE_SIZE_BITS).into()
    }
    // 返回标志位
    pub fn flags(&self) -> PTEFlags {
        //这里只需要标志位，需要把非标志位的位置清零
        let mut bits = self.bits;
        bits.set_bits(14..PALEN, 0);
        PTEFlags::from_bits(bits).unwrap()
    }
    // 有效位
    pub fn is_valid(&self) -> bool {
        (self.flags() & PTEFlags::V) != PTEFlags::empty()
    }
    // 是否可写
    pub fn writable(&self) -> bool {
        (self.flags() & PTEFlags::W) != PTEFlags::empty()
    }
    // 是否可读
    pub fn readable(&self) -> bool {
        !((self.flags() & PTEFlags::NR) != PTEFlags::empty())
    }
    // 是否可执行
    pub fn executable(&self) -> bool {
        !((self.flags() & PTEFlags::NX) != PTEFlags::empty())
    }
    //设置脏位
    pub fn set_dirty(&mut self) {
        self.bits.set_bit(1, true);
    }
    // 用于判断存放的页目录项是否为0
    // 由于页目录项只保存下一级目录的基地址
    // 因此判断是否是有效的就只需判断是否为0即可
    pub fn is_zero(&self) -> bool {
        self.bits == 0
    }
}
```

除了根据标志位定义差异而进行修改的函数，可以看到上述还多出来几个函数，这几个函数主要用于后续构建页表时的遍历过程，与risc-v不同，loongarch下三级页表的构建中第1, 2级页表的页表项并不是前文提到的页表项，而是一个物理地址，其直接保存下一级页表的基地址，只有最后一级页表的页表项存储上述所说明的页表项。

物理页帧的分配方法与rcore相同，没有额外的修改。

在内核中访问一个物理页帧的方法也需要做修改，因为页大小的差异导致了每一页上的页表项数量和字节数组大小都发生了改变

```
impl PhysPageNum {
    pub fn get_pte_array(&self) -> &'static mut [PageTableEntry] {
        let pa: PhysAddr = self.clone().into();
        unsafe { core::slice::from_raw_parts_mut(pa.0 as *mut PageTableEntry,
2048) }
            //每一个页有2048个页表项
    }
    pub fn get_bytes_array(&self) -> &'static mut [u8] {
        let pa: PhysAddr = self.clone().into();
        unsafe { core::slice::from_raw_parts_mut(pa.0 as *mut u8, 16 * 1024) }
    }
}
```

为了取出虚拟地址的三级页表索引，并按照从高到低的顺序返回，也根据页表大小定义做了修改：

```
impl VirtPageNum {
    pub fn indexes(&self) -> [usize; 3] {
        let mut vpn = self.0;
        let mut idx = [0usize; 3];
        for i in (0..3).rev() {
            idx[i] = vpn & 2047; //2^11-1 每一页包含2048个页表项
            vpn >>= 11;
        }
        idx
    }
}
```

```

fn find_pte_create(&mut self, vpn: VirtPageNum) -> Option<&mut PageTableEntry>
{
    let idxs = vpn.indexes();
    let mut ppn = self.root_ppn;
    let mut result: Option<&mut PageTableEntry> = None;
    for i in 0..3 {
        let pte = &mut ppn.get_pte_array()[idxs[i]];
        if i == 2 {
            //找到叶子节点，叶子节点的页表项是否合法由调用者来处理
            result = Some(pte);
            break;
        }
        if pte.is_zero() {
            let frame = frame_alloc().unwrap();
            // 页目录项只保存地址
            *pte = PageTableEntry {
                bits: frame.ppn.0 << PAGE_SIZE_BITS, //存储下一级页表的基地址
            };
            self.frames.push(frame);
        }
        ppn = pte.directory_ppn();
    }
    result
}

pub fn find_pte(&self, vpn: VirtPageNum) -> Option<&mut PageTableEntry> {
    let idxs = vpn.indexes();
    let mut ppn = self.root_ppn;
    let mut result: Option<&mut PageTableEntry> = None;
    for i in 0..3 {
        let pte = &mut ppn.get_pte_array()[idxs[i]];
        if pte.is_zero() {
            return None;
        }
        if i == 2 {
            result = Some(pte);
            break;
        }
        ppn = pte.directory_ppn();
    }
    result
}

```

`PageTable::find_pte` 与 `find_pte_create` 实现中使用了上面额外定义的函数，在申请一个物理页帧的时候，我们会将其清空，因此判断页目录项中是否保存一个有效地址的方法就是直接判断其是否为0。

```

pub fn map(&mut self, vpn: VirtPageNum, ppn: PhysPageNum, flags: PTEFlags) {
    let pte = self.find_pte_create(vpn).unwrap();
    assert!(!pte.is_valid(), "vpn {:?} is mapped before mapping", vpn);
    *pte = PageTableEntry::new(ppn, flags | PTEFlags::default());
}

```

在插入页表项时，需要将标志位与默认的设置进行或操作，因为外部传入的标志位可能并不会正确设置这些默认位。

地址空间

在rcore中，在开启页表机制后，内核代码与应用程序代码均需要通过地址转换，因此在开启页表前需要为内核构建好地址空间。但是在loongarch平台上，有了前文所述的直接映射窗口机制，就可以免去构建内核地址空间的工作，只为用户程序构建地址空间。

在前面的实验中，内核与应用程序使用着BIOS提供的便利，直接访问着物理内存，因此应用程序就可以无视限制直接修改内存的内容，在这一章中，我们需要充分利用loongarch提供的直接映射窗口机制和页表对应用程序的内存访问做出限制，并且降低构建内核地址空间和应用地址空间的难度。

首先，内核地址空间由映射窗口完成映射，而不像rcore中建立恒等映射，但两者所起到的作用是一样的，即访问的虚拟地址就等于物理地址，因此，相对于前面的章节，本章在进入main函数前，需要设置直接映射窗口，实现如下：

```
.section .text.init
.global _start

_start:
0:
    #设置映射窗口
    addi.d $t0,$zero,0x11
    csrwr $t0,0x180  #设置LOONGARCH_CSR_DMWIN0

    la.global $t0,1f
    jirl $zero, $t0,0
1:
    la.global $t0, sbss
    la.global $t1, ebss
    bgeu $t0, $t1, 3f    #bge如果前者大于等于后者则跳转
2:
    st.d $zero, $t0,0
    addi.d $t0, $t0, 8
    bltu $t0, $t1, 2b
3:
    bl main
```

上述汇编代码将0x11写入DMW0寄存器，那么0x0 - 0xFFFFFFFFFFFF的地址范围内，内核所在的特权级PLV0就可以对其进行任意的访问，在链接脚本中，我们指定了内核起始地址为 0x1000，在qemu模拟的地址空间中，0x0-0xffffffff范围为低256MB物理内存，本实验中也会只使用此范围的物理内存。并且上述汇编也将bss段进行了初始化，在跳转到main函数后，内核代码就可以正常执行了。对于应用程序而言，由于映射窗口并没有设置特权级PLV3，而且此时开启了页表机制，因此如果从内核进入应用程序后，由于应用程序的地址空间不能匹配映射窗口，那么只能去查找TLB，而此时TLB中并不会包含如何信息，此时就会发生TLB重填例外，由我们将应用程序地址空间中对应页表项写入TLB。

有了直接映射窗口的实现，逻辑段和地址空间的抽象也需要做出相应的修改

```

pub struct MapArea {
    vpn_range: VPNRange,
    data_frames: BTreeMap<VirtPageNum, FrameTracker>,
    map_perm: MapPermission,
}

```

在逻辑段的定义中删除了map_type字段，因为此时不再区分恒等映射和非恒等映射，逻辑段只有应用程序的非恒等映射。

```

// PTEFlags 的一个子集
// 主要含有几个读写标志位和存在位
bitflags! {
    pub struct MapPermission: usize {
        const NX = 1 << 62;
        const NR = 1 << 61;
        const W = 1 << 8;
        const PLVL = 1 << 2;
        const PLVH = 1 << 3;
        const RPLV = 1 << 63;
    }
}
impl Default for MapPermission {
    fn default() -> Self {
        MapPermission::PLVL | MapPermission::PLVH
    }
}

```

这里MapPermission设置为PTEFlags的一个子集，主要控制逻辑段的读写执行属性一级期望可以运行的特权级。其默认的实现需要设置访问的特权级为3。在 `from_elf` 映射应用程序地址空间时，一般不对RPLV设置，这样一来，在特权级1-3上也可以访问对应的页表项。

```

pub fn insert_area(&mut self, start_va: VirtAddr, end_va: VirtAddr, permission: MapPermission) {
    self.push(MapArea::new(start_va, end_va, permission), None);
}
pub fn map_one(&mut self, page_table: &mut PageTable, vpn: VirtPageNum) {
    let ppn: PhysPageNum;
    let frame = frame_alloc().unwrap();
    ppn = frame.ppn;
    self.data_frames.insert(vpn, frame);
    let pte_flags = PTEFlags::from_bits(self.map_perm.bits).unwrap();
    page_table.map(vpn, ppn, pte_flags);
}
#[allow(unused)]
pub fn unmap_one(&mut self, page_table: &mut PageTable, vpn: VirtPageNum) {
    self.data_frames.remove(&vpn);
    page_table.unmap(vpn);
}

```

由于去掉了多余的映射方式，在实际进行映射和解映射时就可以删除掉多余的判断。`new_kernel` 的实现也被从代码中删除。

```
pub fn from_elf(elf_data: &[u8]) -> (Self, usize, usize) {
    let mut memory_set = Self::new_bare();
    // map program headers of elf, with U flag
    let elf = xmas_elf::ElfFile::new(elf_data).unwrap();
    let elf_header = elf.header;
    let magic = elf_header.pt1.magic;
    assert_eq!(magic, [0x7f, 0x45, 0x4c, 0x46], "invalid elf!");
    let ph_count = elf_header.pt2.ph_count();
    let mut max_end_vpn = VirtPageNum(0);
    for i in 0..ph_count {
        let ph = elf.program_header(i).unwrap();
        if ph.get_type().unwrap() == xmas_elf::program::Type::Load {
            let start_va: VirtAddr = (ph.virtual_addr() as usize).into();
            let end_va: VirtAddr = ((ph.virtual_addr() + ph.mem_size()) as
usize).into();
            let mut map_perm = MapPermission::default();
            let ph_flags = ph.flags();
            if !ph_flags.is_read() {
                map_perm |= MapPermission::NR;
            }
            if ph_flags.is_write() {
                map_perm |= MapPermission::W;
            } //可写
            if !ph_flags.is_execute() {
                map_perm |= MapPermission::NX;
            }
            let map_area = MapArea::new(start_va, end_va, map_perm);
            max_end_vpn = map_area.vpn_range.get_end();
            memory_set.push(
                map_area,
                Some(&elf.input[ph.offset() as usize..(ph.offset() +
ph.file_size()) as usize]),
            );
        }
    }
    // map user stack with U flags
    let max_end_va: VirtAddr = max_end_vpn.into();
    let mut user_stack_bottom: usize = max_end_va.into();
    // guard page
    user_stack_bottom += PAGE_SIZE; //用户栈
    let user_stack_top = user_stack_bottom + USER_STACK_SIZE;
    memory_set.push(
        MapArea::new(
            user_stack_bottom.into(),
            user_stack_top.into(),
            MapPermission::default() | MapPermission::W,
        ),
        None,
    );
    //返回地址空间,用户栈顶,入口地址
    (
        memory_set,
        user_stack_top,
        elf.header.pt2.entry_point() as usize,
    )
}
```

在映射应用程序地址空间的实现中，主要的差异是MapPermission的设置以及比较重要的跳板页和Trap页映射，可以看到，上文中并没有映射这两个页。对于rcore来说，由于地址空间的切换需要在进行trap上下文保存和恢复时进行，而如果将trap上下文保存到内核栈上的话，只有一个sscratch寄存器将无法进行周转，因此其实现中为了保证在切换地址空间时保证指令执行的连续在应用程序地址空间和内核地址空间的最高部分设置了一个跳板页，并且将trap上下文保存在了应用程序的一个虚拟页面中。而在loongarch上，由于切换到内核地址空间并不需要设置相关寄存器的值，通过CRMD特权级PLV字段的变化，应用程序和内核的地址空间将会自动切换，而且就算没有直接映射窗口的支持，loongarch中可以周转的寄存器也可能有多个，这时仍然不需要像rcore一样为了寄存器进行取舍，因此在实现此部分的时候，仍然可以跟前面的实现一样，使用一个寄存器来切换应用程序内核态和用户态栈，并将trap上下文保存在内核栈上，从而我们不需要增加跳板页和trap页的映射。但这也会带来一个比较棘手的问题，那就是应用程序的内核栈由于处于内核地址空间，通过直接映射窗口我们无法设置其权限，也无法设置保护页，这在发生栈溢出时可能就会造成内核代码被破坏的情况，这里暂时也没有较好的方法解决。

通过上述的介绍，我们就可以理解loongarch代码中trap上下文的定义和保存恢复这些代码并没有发生太大的改变，对于内核栈的分配，本章节仍然沿用了前面章节中定义的全局变量

```
static KERNEL_STACK: [KernelStack; MAX_APP_NUM] = [KernelStack {
    data: [0; KERNEL_STACK_SIZE],
}; MAX_APP_NUM];

#[repr(align(4096))]
#[derive(Copy, Clone)]
struct KernelStack {
    data: [u8; KERNEL_STACK_SIZE],
}
```

在任务控制块的定义中，添加了额外的字段

```
pub struct TaskControlBlock {
    pub task_status: TaskStatus,
    pub task_cx_ptr: TaskContext, //任务上下文栈顶地址
    pub memory_set: MemorySet, //新增的地址空间
    pub task_id: usize, //任务id
    pub base_size: usize,
}
```

因为我们会启用ASID功能，这个功能在前文介绍硬件机制有详细说明，因此需要为每个进程设置唯一的编号，这个时候没有进程的概念则使用task_id代替。

任务控制块的建立也变得异常简单：

```

pub fn new(elf_data: &[u8], app_id: usize) -> Self {
    // memory_set with elf program headers/trampoline/trap context/user
    stack
    let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
    let task_status = TaskStatus::Ready; //准备指向状态
    let task_control_block = Self {
        task_status,
        task_cx_ptr: TaskContext::goto_restore(init_app(cx(app_id,
entry_point, user_sp))),
        //初始化任务上下文,参数为内核栈地址, 内核栈存放的是trap上下文
        memory_set,
        task_id: app_id,
        stride: 0,
        pass: 0,
        base_size: user_sp,
    };
    // prepare TrapContext in user space
    task_control_block
}
}

```

在 `init_app(cx)` 的实现中会将 `task_id` 传入。

```

pub fn init_app(cx(app: usize, entry: usize, user_stack_ptr: usize) -> usize {
    //返回任务trap的上下文
    let t = KERNEL_STACK[app].push_context(
        //压入trap上下文
        TrapContext::app_init_context(entry, user_stack_ptr),
    );
    t
}

```

有了上述软件的支持，下面就需要开启页表的相关功能了，使能页表的任务从进入内核就已经完成，我们需要做的是根据前面的文章所属设置好页表相关的寄存器并编写TLB重填异常的代码。

首先是配置页表相关寄存器：

```

pub fn init() {
    extern "C" {
        fn __alltraps();
        fn __tlb_rfill();
        fn kernel_trap_entry();
    }
    Tcfg::read().set_enable(false).write();
    Ecfg::read().set_lie_with_index(11, false).write();
    Crmd::read().set_ie(false).write(); //关闭全局中断
    Entry::read().set_eentry(__alltraps as usize).write(); //设置普通异常和中断入口
}

//设置TLB重填异常地址
TLBREEntry::read()
    .set_val(__tlb_rfill as usize).get_bits(0..32))
    .write(); //设置TLB重填异常入口
SLtbPs::read().set_page_size(0xe).write(); //设置STLB的页面大小为16KiB
TlbREhi::read().set_page_size(0xe).write(); //设置STLB的页面大小为16KiB

Pwcl::read()
    .set_ptbase(0xe)
    .set_ptwidth(0xb)
    .set_dir1_base(25) //页目录表起始位置
    .set_dir1_width(0xb) //页目录表宽度为11位
    .write(); //16KiB的页大小
Pwch::read()
    .set_dir3_base(36) //第三级页目录表
    .set_dir3_width(0xb) //页目录表宽度为11位
    .write();
unsafe {
    asm!("invtlb 0,$r0,$r0"); //清除TLB
}
}

```

- 首先我们设置了TLB重填异常地址的入口为`_tlb_rfill`,其实现稍后给出,在跳转应用程序地址空间后会发生地址转换,这项工作由软件执行,因此代码需要完成将页表项放入TLB的工作
- 设置页大小,由于loongarch下由两个部分构成,我们实验中的页大小都为16kb,因此需要设置寄存器SLTBPS的页大小为16kb
- 设置TLBREHI的PS字段为14,这个位置是TLB重填例外专用的页大小值,在发生TLB重填例外时,执行TLBWR和TLBFILL指令,写入的TLB表项的PS域的值来自于此。
- 设置多级页表各级页表的结构
- 将原来TLB中所有的页表项都无效掉。

TLB重填异常

当TLB重填异常发生后,其异常处理程序的主要处理流程是根据CSR.TLBRBADV中VAddr域记录的虚地址信息以及从CSR.PGD中得到的页目录表PGD的基址信息,遍历发生TLB重填异常的进程的多级页表,从内存中取回页表项信息填入CSR.TLBRELO0和CSR.TLBRELO1的相应域中,最终用TLBFILL指令将页表项填入TLB。前面在讲述TLBFILL指令写操作过程时,提到此时写入TLB的信息除了来自CSR.TLBRELO0和CSR.TLBRELO1的各个域之外,还有来自CSR.ASID中

ASID 域和 CSR.TLBREHI 中 VPPN域的信息。在 TLB 重填异常从发生到进行处理的过程中，软硬件都没有修改 CSR.ASID 中的 ASID 域，所以在执行 TLBFILL 指令时，CSR.ASID 中的 ASID 域记录的就是发生 TLB 重填异常的进程对应的 ASID。至于 CSR.TLBREHI 中的 VPPN 域，在 TLB 重填异常发生并进入异常入口时，已经被硬件填入了触发该异常的虚地址中的虚双页号信息。

整个 TLB 重填异常处理过程中，遍历多级页表是一个较为复杂的操作，需要数十条普通访存、运算指令才能完成，而且如果遍历的页表级数增加，则需要更多的指令。LoongArch 指令系统中定义了 LDDIR 和 LDPTE 指令以及与之配套的 CSR.PWCL 和 CSR.PWCH 来加速 TLB 重填异常处理中的页表遍历。LDDIR 和 LDPTE 指令的功能简述如下图

指令	描述
LDDIR rd, rj, level	将 rj 寄存器中的值作为当前页目录表的基地址，同时根据 CSR.TLBREBADV 中 VAddr 域存放的 TLB 缺失地址以及 PWCL、PWCH 寄存器中定义的页目录表 level 索引的起始位置和位宽信息计算出当前目录页表的偏移量，两者相加作为访存地址，从内存中读取待访问页目录表/页表的基址，写入 rd 寄存器中。
LDPTE rj, seq	将 rj 寄存器中的值作为末级页表的基地址，同时根据 CSR.TLBREBADV 中 VAddr 域存放的 TLB 缺失地址以及 PWCL、PWCH 寄存器中定义的末级页表索引的起始位置和位宽信息计算出末级页表的偏移量，两者相加作为访存地址，从内存中读取偶数号 (seq=0) 或奇数号 (seq=1) 页表项的内容，将其写入到 TLBRELO0 或 TLBRELO1 寄存器中。

CSR.PWCL 和 CSR.PWCH 用来配置 LDDIR 和 LDPTE 指令所遍历页表的规格参数信息，其中 CSR.PWCL 中定义了每个页表项的宽度 (PTEwidth 域) 以及末级页表索引的起始位置和位宽 (PTbase 和 PTwidth 域) 、页目录表 1 索引的起始位置和位宽 (Dir1_base 和 Dir1_width 域) 、页目录表 2 索引的起始位置和位宽 (Dir2_base 和 Dir2_width 域) ,CSR.PWCH 中定义了页目录表 3 索引的起始位置和位宽 (Dir3_base 和 Dir3_width 域) 、页目录表 4 索引的起始位置和位宽 (Dir4_base 和 Dir4_width 域) 。在 loongArch64 中，当进行三级页表的遍历时，通常用 Dir1_base 和 Dir1_width 域来配置页目录表 PMD 索引的起始位置和位宽，用 Dir3_base 和 Dir3_width 域来配置页目录表 PGD 索引的起始位置和位宽，Dir2_base 和 Dir2_width 域、Dir4_base 和 Dir4_width 域空闲不用。

使用上述指令，TLB 重填异常处理程序如下：

```

.section tlb_handler
.globl __tlb_rfill
.align 4
__tlb_rfill:
    csrwr $t0, 0x8B    #保存t0的值到CSR_TLBRSAVE
    csrrd $t0, 0x1B    #读取PGD,类似于rcore中的token
    lddir $t0, $t0, 3  #访问页目录表PGD
    lddir $t0, $t0, 1  #访问页目录表PMD
    ldpte $t0, 0
    #取回偶数号页表项
    ldpte $t0, 1
    #取回奇数号页表项
    tlbfill
    csrrd $t0, 0x8B
    #jr $ra
    ertn

```

重填异常的处理中需要获取地址空间的根页表地址，类似rcore中token，但这里是完整的物理地址，而不是物理页号，所以在memoryset的实现中也可以看到返回token的实现存在差异。完成TLB重填的任务，还需要解决的是页修改例外，在risc-v上页表的访问由硬件完成，因此某些位将由硬件来设置，但loongarch上全部由软件完成，包括其中的D位，这是表示该页表项对应的数据是否有被写过，在构建页表项时，这个位被设置为0，当程序发生写操作时，如果这位为0会发生页修改例外，而我们需要做的就是修改应用地址空间页表项的对应位以及修改TLB中的页表项对应位。其实现如下：

```

/// Exception(PageModifyFault)的处理
/// 页修改例外: store 操作的虚地址在 TLB 中找到了匹配, 且 V=1, 且特权等级合规的项, 但是该页
// 表项的 D 位为 0, 将触发该例外
fn tlb_page_modify_handler() {
    // INFO!("PageModifyFault handler");
    //找到对应的页表项, 修改D位为1
    let badv = TlbRBadv::read().get_val(); //出错虚拟地址
    let vpn: VirtAddr = badv.into(); //虚拟地址
    let vpn: VirtPageNum = vpn.floor(); //虚拟地址的虚拟页号
    let token = current_user_token(); //根页表的地址
    let page_table = PageTable::from_token(token);
    let pte = page_table.find_pte(vpn).unwrap(); //获取页表项
    pte.set_dirty(); //修改D位为1
    unsafe {
        asm!("tlbsrch", "tlbrd",); //根据TLBEHI的虚双页号查询TLB对应项
    }
    let tlbidx = TlbIdx::read(); //获取TLB项索引
    assert_eq!(tlbidx.get_ne(), false);
    let mut tlbelo0 = TLBEL0::read(0); //获取TLB项0
    let mut tlbelo1 = TLBEL0::read(1); //获取TLB项1
    tlbelo0.set_dirty(true).write();
    tlbelo1.set_dirty(true).write();
    unsafe {
        asm!("tlbwr"); //重新将tlbelo写入tlb
    }
}

```

在完成上述工作后页表机制就可以正常运行了。在开启时钟的实现部分，我们修改了相关实现：

```
pub fn enable_timer_interrupt() {
    let timer_freq = get_timer_freq();
    Ecfg::read().set_lie_with_index(11, true).write();
    Ticlr::read().clear_timer().write(); //清除时钟专断
    Tcfg::read()
        .set_enable(true)
        .set_loop(true)
        .set_tval(timer_freq / TICKS_PER_SEC)
        .write(); //设置计时器的配置

    Crmd::read().set_ie(true).write(); //开启全局中断
}
```

在实验前期时钟的中断间隔被我们设置了一个大概值，这里通过读取cpu配置字查询到了时钟的频率，可以正确配置周期间隔了。

其中trap的处理也做了对应于rcore的修改

```
pub fn trap_return(){
    set_user_trap_entry();
    let trap_cx = current_trap(cx);
    extern "C"{
        fn __restore();
    }
    unsafe{
        asm!("move $a0,{}", in(reg)trap_cx);
        __restore();
    }
}
```

这里要简单的多，只需要设置用户态发生异常时的入口然后将用户程序的trap上下文所在内核栈的地址传入a0即可。

在切换任务时，处理切换任务上下文，我们还需要切换地址空间和设置ASID

```

fn run_next_task(&self) {
    if let Some(next) = self.find_next_task() {
        //查询是否有处于准备的任务，如果有就运行
        let mut inner = self.inner.borrow();
        let current_task = inner.current_task;
        inner.current_task = next;
        inner.tasks[next].task_status = TaskStatus::Running;
        //获取两个任务的task上下文指针
        let current_task_cx_ptr =
            &mut inner.tasks[current_task].task_cx_ptr as *mut TaskContext;
        let next_task_cx_ptr2 = &inner.tasks[next].task_cx_ptr as *const TaskContext;
        let pgd = inner.tasks[next].get_user_token() << PAGE_SIZE_BITS; //
        获得根页表基地址
        Pgdl::read().set_val(pgd).write(); //设置根页表基地址
        let current_task_id = inner.tasks[next].task_id;
        //释放可变借用，否则进入下一个任务后将不能获取到inner的使用权
        drop(inner);
        unsafe {
            __switch(current_task_cx_ptr, next_task_cx_ptr2,
current_task_id);
        }
    } else {
        panic!("There are no tasks!");
    }
}

```

在代码中我们获取了当前任务的根页表物理页号并将其转为物理地址写入PGDL寄存器中，在`__switch`中，传入了要切换到的任务id，在汇编代码实现中，会将id写入ASID寄存器，这样一来，当任务回复trap上下文回到用户态后，就会根据这两个寄存器来进行地址转换了。

第五章

第五章开始引入进程的抽象，在前文任务的基础上可以更加合理有效地管理资源，这一章主要涉及软件层面的设计，因此可以不用关注太多的硬件细节。主要涉及的机器差异可能来自于在何时无效TLB的内容。

- 内核栈的设计
- fork和exec的差异
- 无效TLB表项

内核栈

在前一章开启地址空间后，应用程序的内核栈我们仍然沿用了以前章节使用的全局分配。在引入进程抽象后，原来的task_id现在变成了进程标识符pid，为了不让内核栈的分配被固定死，这一章需要重新设计内核栈的分配，在rcore中是根据pid从地址空间高位依次分配，而本实验则选择了直接分配分配物理页帧

```
// Kernelstack for app
#[derive(Clone, Debug)]
pub struct KernelStack {
    frame: FrameTracker,
}
```

内核栈定义上所示，其只包含一个物理页帧，当然这里为内核栈分配16kb的空间可能会比较浪费，但为了简单起见，这里就直接设置如此。在内核栈上，我们需要保存应用程序的trap上下文，而且在进行fork时也需要拷贝父进程的trap上下文，所以其需要实现一些方便的接口：

```

pub fn new() -> Self {
    Self {
        frame: frame_alloc().unwrap(),
    }
}

pub fn copy_from_other(&mut self, kernel_stack: &KernelStack) -> &mut Self
{
    //需要从kernel_stack复制到self
    let trap_context = kernel_stack.get_trap_cx().clone();
    self.push_on_top(trap_context);
    self
}

#[allow(unused)]
///Push a value on top of kernelstack
pub fn push_on_top<T>(&self, value: T) -> *mut T
where
    T: Sized,
{
    let kernel_stack_top = self.get_top();
    let ptr_mut = (kernel_stack_top - core::mem::size_of::<T>()) as *mut T;
    unsafe {
        *ptr_mut = value;
    }
    ptr_mut
}

///Get the value on the top of kernelstack
pub fn get_top(&self) -> usize {
    let top :PhysAddr= self.frame.ppn.into();
    let top = top.0 + PAGE_SIZE;
    top
}

/// 返回trap上下文的可变引用
/// 用于修改返回值
pub fn get_trap_cx(&self) -> &'static mut TrapContext {
    let cx = self.get_top() - core::mem::size_of::<TrapContext>();
    unsafe { &mut *(cx as *mut TrapContext) }
}

/// 返回trap上下文的位置，用于初始化trap上下文
pub fn get_trap_addr(&self) -> usize {
    let addr = self.get_top() - core::mem::size_of::<TrapContext>();
    addr
}

```

进程控制块的定义如下：

```
pub struct TaskControlBlock {
    // immutable
    pub pid: PidHandle,
    // mutable
    inner: UPSafeCell<TaskControlBlockInner>,
}

pub struct TaskControlBlockInner {
    pub kernel_stack: KernelStack,
    pub base_size: usize,
    pub task_cx: TaskContext, //任务上下文栈顶地址
    pub task_status: TaskStatus,
    pub memory_set: MemorySet, //新增的地址空间
    pub parent: Option<Weak<TaskControlBlock>>,
    pub children: Vec<Arc<TaskControlBlock>>,
    pub exit_code: i32,
}

impl TaskControlBlockInner {
    pub fn get_trap_cx(&self) -> &'static mut TrapContext {
        self.kernel_stack.get_trap_cx()
    }
}
```

差别之处主要在于应用程序的trap上下文此时会从内核栈上直接获取，而不需要去查询页表到应用程序地址空间中查找。

在进程调度中，此时有了pid的存在，原来的task_id页需要被修改，同时，也将其从__switch中移到外部。

```
pub fn run_tasks() {
    loop {
        let mut processor = PROCESSOR.exclusive_access();
        if let Some(task) = fetch_task() {
            let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
            // access coming task TCB exclusively
            let mut task_inner = task.inner_exclusive_access();
            let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
            task_inner.task_status = TaskStatus::Running;

            let pid = task.getpid(); //应用进程号
            let pgd = task_inner.get_user_token() << PAGE_SIZE_BITS;
            Pgdl::read().set_val(pgd).write(); //设置根页表基地址
            Asid::read().set_asid(pid as u32).write(); //设置ASID

            drop(task_inner);
            // release coming task TCB manually
            processor.current = Some(task);
            // release processor manually
            drop(processor);
            unsafe {
                __switch(idle_task_cx_ptr, next_task_cx_ptr);
            }
        }
    }
}
```

上述代码的中间部分获取进程pid，然后获取根页表将其转换为物理地址，分别将其写入PGDL和ASID寄存器中。

fork和exec

进程的创建：

```
pub fn new(elf_data: &[u8]) -> Self {
    // memory_set with elf program headers/trampoline/trap context/user stack
    let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
    let task_status = TaskStatus::Ready; //准备状态
    let pid = pid_alloc(); //分配pid

    let kernel_stack = KernelStack::new(); //分配内核栈

    let kernel_trap_cx =
        kernel_stack.push_on_top(TrapContext::app_init_context(entry_point,
user_sp));
    //在内核栈放入trap上下文
    let task_control_block = Self {
        pid,
        inner: unsafe {
            UPSafeCell::new(TaskControlBlockInner {
                kernel_stack,
                base_size: 0,
                task_cx: TaskContext::goto_restore(kernel_trap_cx as usize),
                //初始化任务上下文，参数为内核栈地址，内核栈存放的是trap上下文
                task_status,
                memory_set,
                parent: None,
                children: Vec::new(),
                exit_code: 0,
                stride: 0,
                pass: 0,
            })
        },
    };
    // prepare TrapContext in user space
    task_control_block
}
```

可以看到，相对于rcore中的实现来说，这里简化了许多，去掉了根据进程号申请内核栈的步骤，而是直接向内核申请内核栈，删除了trap页的初始化，此时trap上下文保存在内核栈顶。

```

pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {
    // ---- access parent PCB exclusively
    let mut parent_inner = self.inner_exclusive_access();
    // copy user space(include trap context)
    let memory_set = MemorySet::from_existed_user(&parent_inner.memory_set);
    // alloc a pid and a kernel stack in kernel space
    let pid_handle = pid_alloc();
    //需要保证子进程与父进程的内核栈信息一样
    let kernel_stack = KernelStack::new();
    let kstack_ptr = kernel_stack.get_trap_addr();
    let task_control_block = Arc::new(TaskControlBlock {
        pid: pid_handle,
        inner: unsafe {
            let inner = UPSafeCell::new(TaskControlBlockInner {
                kernel_stack,
                base_size: parent_inner.base_size,
                task_cx: TaskContext::goto_restore(kstack_ptr),
                task_status: TaskStatus::Ready,
                memory_set,
                parent: Some(Arc::downgrade(self)),
                children: Vec::new(),
                exit_code: 0,
                stride: 0,
                pass: 0,
            });
            inner
        },
    });
    // add child
    task_control_block
        .inner_exclusive_access()
        .kernel_stack
        .copy_from_other(&parent_inner.kernel_stack);
    parent_inner.children.push(task_control_block.clone());
    task_control_block
}

```

在 `fork` 的实现中，由于在复制父进程地址空间时不会进行内核栈和trap页的映射，因此这里就需要将父进程的trap上下文直接复制到子进程的内核栈中从而可以让两者回到用户态的瞬间处于同一状态。

```

pub fn exec(&self, elf_data: &[u8]) {
    // memory_set with elf program headers/trampoline/trap context/user
    stack
    let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
    // **** access inner exclusively
    let mut inner = self.inner_exclusive_access();
    // substitute memory_set
    inner.memory_set = memory_set; //覆盖 memory_set
                                    // initialize trap_cx
    inner
        .kernel_stack
        .push_on_top(TrapContext::app_init_context(entry_point, user_sp));
    //由于切换了地址空间，因此之前的ASID对应的地址空间将不会再有用，因此这里需要将TLB中
    的内容无效掉
    let pid = self.getpid();
    unsafe {
        asm!("invtlb 0x4,{},$r0", in(reg) pid);
    }
    let pgd = inner.get_user_token() << PAGE_SIZE_BITS;
    Pgdl::read().set_val(pgd).write();
    // **** release inner automatically
}

```

exec 的实现中，差异主要来自于最后我们加入了TLB无效指令并且重新设置了根页表的地址。

指令格式： invtlb op, rj, rk

INVTLB 指令用于无效 TLB 中的内容。指令的三个源操作数中，op 是 5 比特立即数，用于指示操作类型。通用寄存器 rj 的[9:0]位存放无效操作所需的 ASID 信息（称为“寄存器指定 ASID”），其余比特必须填 0。当 op 所指示的操作不需要 ASID 时，应将通用寄存器 rj 设置为 r0。通用寄存器 rk 中用于存放无效操作所需的虚拟地址信息（称为“寄存器指定 VA”）。当 op 所指示的操作不需要虚拟地址信息时，应将通用寄存器 rk 设置为 r0

op	操作
0x0	清除所有页表项。
0x1	清除所有页表项。此时操作效果与 op=0 完全一致。
0x2	清除所有 G=1 的页表项。
0x3	清除所有 G=0 的页表项。
0x4	清除所有 G=0，且 ASID 等于寄存器指定 ASID 的页表项。
0x5	清除 G=0，且 ASID 等于寄存器指定 ASID，且 VA 等于寄存器指定 VA 的页表项。
0x6	清除所有 G=1 或 ASID 等于寄存器指定 ASID，且 VA 等于寄存器指定 VA 的页表项。

在fork完成后，子进程与父进程的地址空间是相同的，在执行exec后，子进程将重新创建地址空间，那么原来的虚拟地址可能在新的地址空间中与原来就不同了，此时就需要将原来已经存在的对应关系删除掉，在开启ASID的情况下，我们只需要根据进程号直接无效掉TLB中的表项即可，而地址空间的改变同时也需要设置根页表对应的寄存器，这样在回到用户态重新进行地址转换时才能正常进行。

在其它函数的实现中，一些需要注意的地方如下：

```
// syscall/process/sys_fork
trap_cx.x[4] = 0; //x[4] is return value
add_task(new_task); // add new task to scheduler
new_pid as isize
```

在fork系统调用时，需要根据loongarch的寄存器做相应的设置

```
// task/mod/exit_current_and_run_next
// 使得原来的TLB表项无效掉，否则下一个进程与当前退出的进程号相同会导致
// 无法正确进行地址转换
unsafe {
asm!("invlrb 0x4,{},$r0",in(reg) pid);
}
```

在一个进程退出时，我们同样需要无效掉其对应的页表项，由于退出时进程的资源会被回收，那么在创建下一个进程时，就可能发生新进程会使用退出进程使用过的物理页，如果TLB中存在快照，就会造成访问错误，因此需要将退出进程在TLB中的快照删除。

第六章

第六章带来了文件系统的实现，由于文件系统是在用户态实现，在接入内核之前都与具体的硬件无关，因此对其实现我们无需修改即可进行使用。在接入到内核的实现中，由于qemu模拟的loongarch平台没有模拟相关的virtio-block设备，因此需要使用其它块设备进行实验，这里选择是SATA设备，rcore的x86版本中有相关的库提供了很好的实现，本实验同样使用了这个库并对硬件相关的内容做了相应的修改。

如果读者有兴趣查看AHCI协议的实现，可阅读文章给出的链接，在后期本教程可能也会将详细的编写过程给出。本章设计的内容包括：

- PCI总线
- PCI设备探测
- STAT设备
- AHCI协议
- DMA

总线

总线（Bus）是指计算机组件间规范化的交换数据（data）的方式，即以一种通用的方式为各组件提供数据传送和控制逻辑。从另一个角度来看，如果说主板（Mother Board）是一座城市，那么总线就像是城市里的公共汽车（bus），能按照固定行车路线，传输来回不停运作的比特（bit）。这些线路在同一时间内都仅能负责传输一个比特。因此，必须同时采用多条线路才能发送更多资料，而总线可同时传输的资料数就称为宽度（width），以比特为单位，总线宽度愈大，传输性能就愈佳。总线的带宽（即单位时间内可以传输的总资料数）为：总线带宽 = 频率×宽度

（Bytes/sec）[wiki](#)。总线含义很广，它不仅仅是指用于数据交换的通道，有时也包含了软件硬件架构。比如 PCI 总线、USB 总线，它们不仅仅是指主板上的某个接口，还包含了与之相对应的整套硬件模型、软件架构。

按照数据传递的方向，总线可以分为单向总线和双向总线。双向总线也称为双工总线。双工总线又可分为半双工总线和全双工总线。半双工总线是指在一个时间段内，数据只能从一个方向传送到另一个方向，数据不能同时在两个方向传递。全双工总线是指数据可以同时在两个方向传递。全双工总线包含两组数据线，分别用于两个方向的数据传输。

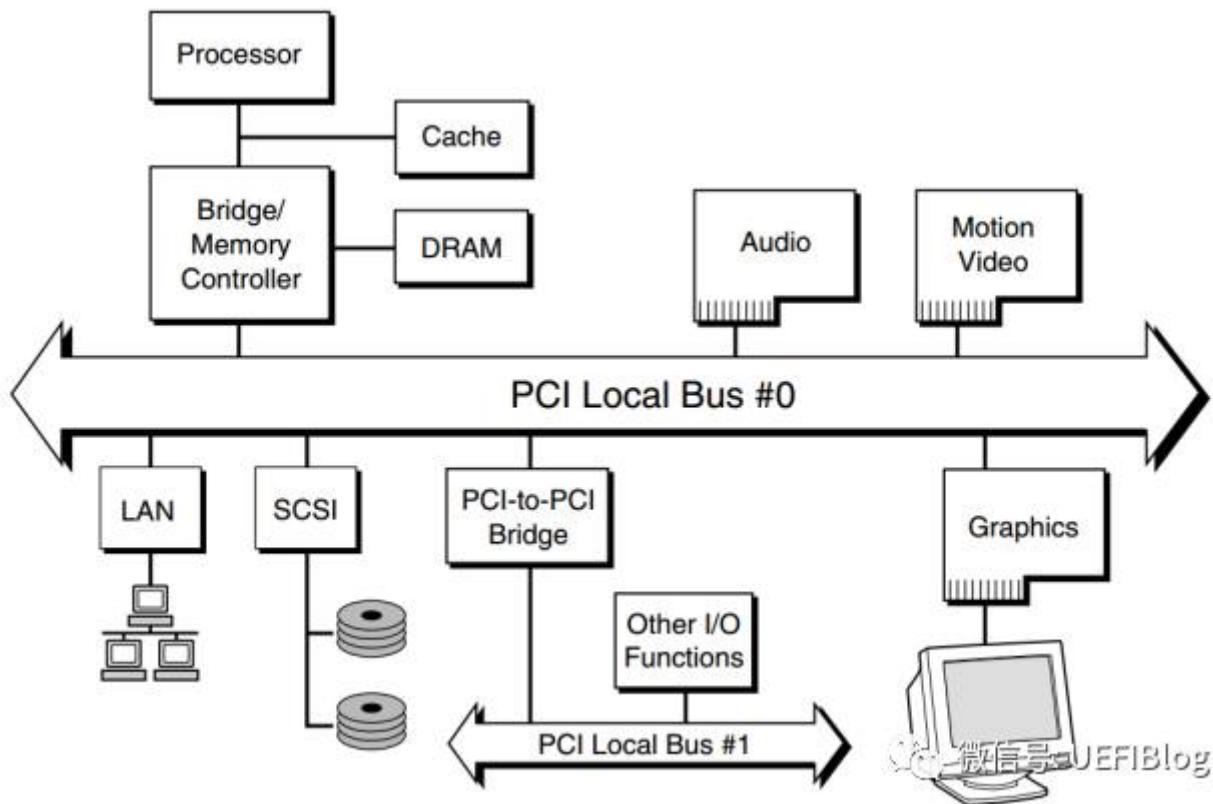
按照总线使用的信号类型，总线可以分为并行总线和串行总线，并行总线包含多位传输线，在同一时刻可以传输多位数据，而串行总线只使用一位传输线，同一时刻只传输一位数据。按照总线在计算机系统中所处的物理位置，总线可以分为片上总线、内存总线、系统总线和设备总线。片上总线是指芯片片内互连使用的总线。芯片在设计时，常常要分为多个功能模块，这些功能模块之间的连接即采用片上互连总线，比如广泛使用的 ARM 公司的 AMBA 系列总线。内存总线用于连接处理器和主存储器，由数据总线与地址总线构成。系统总线通常用于处理器与桥片的连接，同时也作为多处理器间的连接以构成多路系统。英特尔处理器所广泛采用的 QPI (Quick PathInterconnect) 接口及在 QPI 之前的 FSB (Front SideBus)，还有 AMD 处理器所广泛采用的 HT (HyperTransport) 接口都属于系统总线。系统总线是处理器与其他芯片进行数据交换的主要通道，系统总线的数据传输能力对计算机整体性能影响很大。如果没有足够带宽的系统总线，计算机系统的外设访问速度会明显受限，类似于显示、存储、网络等设备的交互都会受到影响。设备总线用于计算机系统中与 IO 设备的连接。PCI (Peripheral Component Interconnect) 总线是一种对计算机体系结构连接影响深远并广泛应用的设备总线。PCIE (PCI Express) 可以被看作 PCI 总线的升级版本，兼容 PCI 软件架构。PCIE 总线被广泛地用作连接设备的通用总线，在现有计算机系统中已经基本取代了 PCI 的位置。

pci设备探测

在rcore中，为了在内核中接入文件系统，添加了块设备到模拟的机器上，但qemu模拟的loongarch机器上似乎无法使用 `virtio-blk-device` 设备，因此我们选择使用STATA硬盘模拟，并添加了Ahci协议。在qemu的启动项中需要添加相应的命令

```
@qemu-system-loongarch64 \
-m 1G \
-smp 1 \
bios $(BOOTLOADER) \
-kernel $(KERNEL_ELF) \
-vga none -nographic \
-drive file=$(FS_IMG),if=none,format=raw,id=x0 \
-device ahci,id=ahci0 \
-device ide-hd,drive=x0,bus=ahci0.0
```

PCI总线的全称是Peripheral Component Interconnect，也就是外围设备互联总线，PCI总线是一种共享总线，总线上的设备分时共享这条总线。其简易的示意图如下所示：



对于OS的开发者来说，我们需要做的就是获取PCI总线的设备，并对相应的设备完成正确的配置。

在PCI协议下，IO的系统空间分为三个部分：配置空间、IO空间和Memory空间。配置空间存储设备的基本信息，主要用于设备的探测和发现；IO空间比较小，用于少量的设备寄存器访问；Memory空间可映射的区域较大，可以方便地映射设备所需要的大块物理地址空间。由于PCI支持设备即插即用，所以PCI设备不占用固定的内存地址空间或I/O地址空间，而是由操作系统决定其映射的基址。系统加电时，BIOS检测PCI总线，确定所有连接在PCI总线上的设备以及它们的配置要

求，并进行系统配置。所以，所有的PCI设备必须实现配置空间，从而能够实现参数的自动配置，实现真正的即插即用。

对于X86架构来说，IO空间的访问需要使用IO指令操作，Memory空间的访问则需要使用通常的load/store指令操作。而对于MIPS或者LoongArch这种把设备和存储空间统一编址的体系结构来说，IO空间和Memory空间没有太大区别，都使用load/store指令操作。IO空间与Memory空间的区别仅在于所在的地址段不同，对于某些设备的Memory访问，可能可以采用更长的单次访问请求。例如对于IO空间，可以限制为仅能使用字访问，而对于Memory空间，则可以任意地使用字、双字甚至更长的Cache行访问。

PCI总线规范定义的每个设备的配置空间总长度为256个字节，配置信息按一定的顺序和大小依次存放。前64个字节的配置空间称为配置头，对于所有的设备都一样，配置头的主要功能是用来识别设备、定义主机访问PCI卡的方式（I/O访问或者存储器访问，还有中断信息）。其余的192个字节称为本地配置空间（设备有关区），主要定义卡上局部总线的特性、本地空间基地址及范围等。每个设备的配置空间中的地址偏移由总线号、设备号、功能号和寄存器号的组合得到，通过对这个组合的全部枚举，可以很方便地检测到系统中存在的所有设备。

Bit 31	Bits 30-24	Bits 23-16	Bits 15-11	Bits 10-8	Bits 7-0
addr	addr	Bus Number	Device Number	Function Number	Register Offset

256字节的寄存器分布如下所示：

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x0	0x0	Device ID		Vendor ID	
0x1	0x4	Status		Command	
0x2	0x8	Class code	Subclass	Prog IF	Revision ID
0x3	0xC	BIST	Header type	Latency Timer	Cache Line Size
0x4	0x10	Base address #0 (BAR0)			
0x5	0x14	Base address #1 (BAR1)			
0x6	0x18	Base address #2 (BAR2)			
0x7	0x1C	Base address #3 (BAR3)			
0x8	0x20	Base address #4 (BAR4)			
0x9	0x24	Base address #5 (BAR5)			
0xA	0x28	Cardbus CIS Pointer			
0xB	0x2C	Subsystem ID		Subsystem Vendor ID	
0xC	0x30	Expansion ROM base address			
0xD	0x34	Reserved			Capabilities Pointer
0xE	0x38	Reserved			
0xF	0x3C	Max latency	Min Grant	Interrupt PIN	Interrupt Line

31	24 23	16 15	8 7	0
设备识别号		厂商识别号		
状态		命令		
类别码			版本号	
BIST	首部类型	延时计时	缓存行大小	
基地址寄存器				
CardBus CIS指针				
子系统号		子系统厂商识别号		
扩展ROM基地址				
保留			寄存器指针	
保留				
最大延迟	最小许可时间	中断引脚	中断线	

厂商识别号 (Vendor ID) 与设备识别号 (Device ID) 的组合是唯一的，由专门的组织进行管理。每一个提供 PCI 设备的厂商都应该拥有唯一的厂商识别号，以在设备枚举时正确地找到其对应的驱动程序。其中 class code 和 subclass 字段合起来可以识别出这个设备的具体类型，比如说块设备或者是网卡设备。

访问设备配置空间有两种方式，x86下通常会使用I/O端口进行访问，在loongarch下我们使用MMIO方式访问，查看qemu的源代码可以知道配置空间的基址为0x2000_0000，我们需要将上述地址偏移+基址才能得到设备配置空间。在配置空间中，并没有设备本身功能上所使用的寄存器。这些寄存器实际上是由可配置的 IO 空间或 Memory 空间来索引的，配置空间中存在 6 组独立的基址寄存器 (Base Address Registers，简称 BAR)。这些 BAR 一方面用于告诉软件该设备所需要的地址空间类型及其大小，另一方面用于接收软件给其配置的基地址。

BAR 的寄存器定义如下图所示，其最低位表示该 BAR 是 IO 空间还是 Memory 空间。BAR 中间有一部分只读位为 0，正是这些 0 的个数表示该 BAR 所映射空间的大小，也就是说 BAR 所映射的空间为 2 的幂次方大小。BAR 的高位是可写位，用来存储软件设置的基地址

31	$n \ n-1$	4	3	2	1	0
可写位	只读0位	可预取标识	64位 标识	IO 标识		

图 7.5: BAR 的寄存器定义

对 PCI 设备的探测和驱动加载是一个递归调用过程，大致算法如下：

1. 将初始总线号、初始设备号、初始功能号设为 0

2. 使用当前的总线号、设备号、功能号组成一个配置空间地址，使用该地址，访问其 0 号寄存器，检查其设备号。
3. 如果读出全 1 或全 0，表示无设备。
4. 如果该设备为有效设备，检查每个 BAR 所需的空间大小，并收集相关信息。
5. 检测其是否为一个多功能设备(Header Type)，如果是则将功能号加 1 再重复扫描，执行第 2 步。
6. 如果该设备为桥设备，则给该桥配置一个新的总线号，再使用该总线号，从设备号 0、功能号 0 开始递归调用，执行第 2 步。
7. 如果设备号非 31，则设备号加 1，继续执行第 2 步；如果设备号为 31，且总线号为 0，表示扫描结束，如果总线号非 0，则退回上一层递归调用。

为了完成PCI设备扫描，我们引入了pci库来简化实现，原来的库使用x86结构下的端口进行访问，因此我们需要将部分实现修改为使用MMIO方式访问

```
#[derive(Debug, Copy, Clone, PartialEq, Eq)]
pub enum CSpaceAccessMethod {
    MemoryMapped,
}
```

在访问配置空间的定义中，我们删除掉了I/O访问方式

```
// Returns a value in native endian.
pub unsafe fn read32<T: PortOps>(self, _ops: &T, loc: Location, offset: u16) -> u32 {
    debug_assert!(
        (offset & 0b11) == 0,
        "misaligned PCI configuration dword u32 read"
    );
    let addr = loc.encode() + (offset as usize);
    match self {
        CSpaceAccessMethod::MemoryMapped => {
            let addr = addr as *const u32;
            addr.read_volatile()
        }
    }
}

pub unsafe fn write32<T: PortOps>(self, _ops: &T, loc: Location, offset: u16, val: u32) {
    debug_assert!(
        (offset & 0b11) == 0,
        "misaligned PCI configuration dword u32 read"
    );
    let addr = loc.encode() + (offset as usize);
    match self {
        CSpaceAccessMethod::MemoryMapped => {
            let addr = addr as *mut u32;
            addr.write_volatile(val)
        }
    }
}
```

在读取寄存器值的部分，直接访问内存而不通过端口。

```

#[derive(Debug, Copy, Clone, PartialEq, Eq)]
pub struct Location {
    base_addr: usize, //base address of the device
    pub bus: u8,
    pub device: u8,
    pub function: u8,
}
impl Location {
    #[inline(always)]
    fn encode(self) -> usize {
        self.base_addr
            | ((self.bus as usize) << 16)
            | ((self.device as usize) << 11)
            | ((self.function as usize) << 8)
    }
}

```

在配置空间偏移地址的定义中，也需要根据MMIO方式加入基地址。根据上文给出的构造方式完成偏移地址的构建。

扫描pci的代码如下：

```

for dev in unsafe {
    scan_bus(
        &UnusedPort,
        CSpaceAccessMethod::MemoryMapped,
        PCI_CONFIG_ADDRESS,
    )
} {
    info!(
        "pci: {:02x}:{:02x}.{:} {:#x} {:#x} ({}) {} irq: {}:{}",
        dev.loc.bus,
        dev.loc.device,
        dev.loc.function,
        dev.id.vendor_id,
        dev.id.device_id,
        dev.id.class,
        dev.id.subclass,
        dev.pic_interrupt_line,
        dev.interrupt_pin
    );
    dev.bars.iter().enumerate().for_each(|(index, bar)| {
        if let Some(BAR::Memory(pa, len, _, t)) = bar {
            info!("\\tbar#{}) (MMIO) {:#x} [{:#x}] [{}]", index, pa, len, t);
        } else if let Some(BAR::IO(pa, len)) = bar {
            info!("\\tbar#{}) (IO) {:#x} [{:#x}]", index, pa, len);
        }
    });
}

```

设备需要通过中断与CPU通讯，在配置空间中的 `interrupt_line` 提供了设备使用的IRQ，但我们不使用这种方法，而是需要启用MSI中断，loongarch的中断机制在后续会进行介绍。在PCI中，MSI

中断是可选功能，因此需要在配置空间的能力链表中查询，能力链表的寄存器如下所示

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
Cap + 0x0	Cap + 0x0	Message Control		Next pointer	Capability ID = 05
Cap + 0x1	Cap + 0x4	Message Address [Low]			
Cap + 0x2	Cap + 0x8	[Message Address High]			
Cap + 0x2/0x3	Cap + 0x8/0xC	Reserved		Message Data	
Cap + 0x4	Cap + 0x10	[Mask]			
Cap + 0x5	Cap + 0x14	[Pending]			

如果低8位寄存器的值为0x5，说明设备支持MSI中断。在知道设备支持MSI中断后下一步就是去启用MSI中断。这需要通过设置上图中message control寄存器，其定义如下：

Bits 15-9	Bit 8	Bit 7	Bits 6-4	Bits 3-1	Bit 0
Reserved	Per-vector masking	64-bit	Multiple Message Enable	Multiple Message Capable	Enable

第0位控制启用MSI中断，第1-3位设置设备最多使用中断的数量，本实验中默认为1，第4-6位设置设备允许使用的中断数量，这里也是1。具体的设置在下节中设置STAT设备有介绍。

块设备驱动

在上小节中完成了PCI设备的探测，文末介绍了设备配置的一点相关内容，在这一节，我们需要争夺SATA设备进行配置，在PCI设备扫描中，当我们探测到了SATA设备时，会进行中断配置

```
if dev.id.class == 0x01 && dev.id.subclass == 0x06 {
    // Mass storage class, SATA subclass
    if let Some(BAR::Memory(pa, len, _, _)) = dev.bars[5] {
        info!("Found AHCI device");
        // 检查status的第五位是否为1，如果是，则说明该设备存在能力链表
        if dev.status | Status::CAPABILITIES_LIST == Status::empty() {
            info!("No capabilities list");
            return None;
        }
        unsafe { enable(dev.loc) };
        assert!((len as usize) < PAGE_SIZE);
        if let Some(x) = AHCIDriver::new(pa as usize, len as usize) {
            return Some(x);
        }
    }
}
```

通过class 和subclass字段的对应值，判断是否找到SATA设备，并检查设备是否存在能力链表，如果存在那么我们将会配置MSI中断

```

unsafe fn enable(loc: Location) {
    let ops = &UnusedPort;
    let am = CSpaceAccessMethod::MemoryMapped;
    // 23 and lower are used
    static mut MSI_IRQ: u32 = 23;

    let orig = am.read16(ops, loc, PCI_COMMAND);
    // bit0      |bit1      |bit2      |bit3      |bit10
    // IO Space |MEM Space |Bus Mastering |Special Cycles |PCI Interrupt
Disable
    am.write32(ops, loc, PCI_COMMAND, (orig | 0x40f) as u32);
    //0100 0000 1111
    // find MSI cap
    let mut msi_found = false;
    let mut cap_ptr = am.read8(ops, loc, PCI_CAP_PTR) as u16; // 能力链表的起始地
址
    while cap_ptr > 0 {
        // info!("cap_ptr: {:#x}", cap_ptr);
        let cap_id = am.read8(ops, loc, cap_ptr);
        if cap_id == PCI_CAP_ID_MSI {
            let orig_ctrl = am.read32(ops, loc, cap_ptr + PCI_MSI_CTRL_CAP);

            // 在 3A+7A 的系统中, PCI MSI 中断的目标地址为 0xfd8000000 或者
0x2ff00000。桥片会将设
            // 备发给这两个地址段的 MSI 消息包, 转换成 HT 中断消息包, 发送给处理器。
            am.write32(ops, loc, cap_ptr + PCI_MSI_ADDR, 0x2ff0_0000); // 设置
MSI的地址
            MSI_IRQ += 1;
            let irq = MSI_IRQ;
            // 检查是64位/32位模式
            // we offset all our irq numbers by 32
            if (orig_ctrl >> 16) & (1 << 7) != 0 {
                // 64bit
                am.write32(ops, loc, cap_ptr + PCI_MSI_DATA_64, irq + 32);
            } else {
                // 32bit
                am.write32(ops, loc, cap_ptr + PCI_MSI_DATA_32, irq + 32);
            }

            // enable MSI interrupt, assuming 64bit for now
            am.write32(ops, loc, cap_ptr + PCI_MSI_CTRL_CAP, orig_ctrl |
0x10000);
            msi_found = true;
        }
        cap_ptr = am.read8(ops, loc, cap_ptr + 1) as u16;
    }

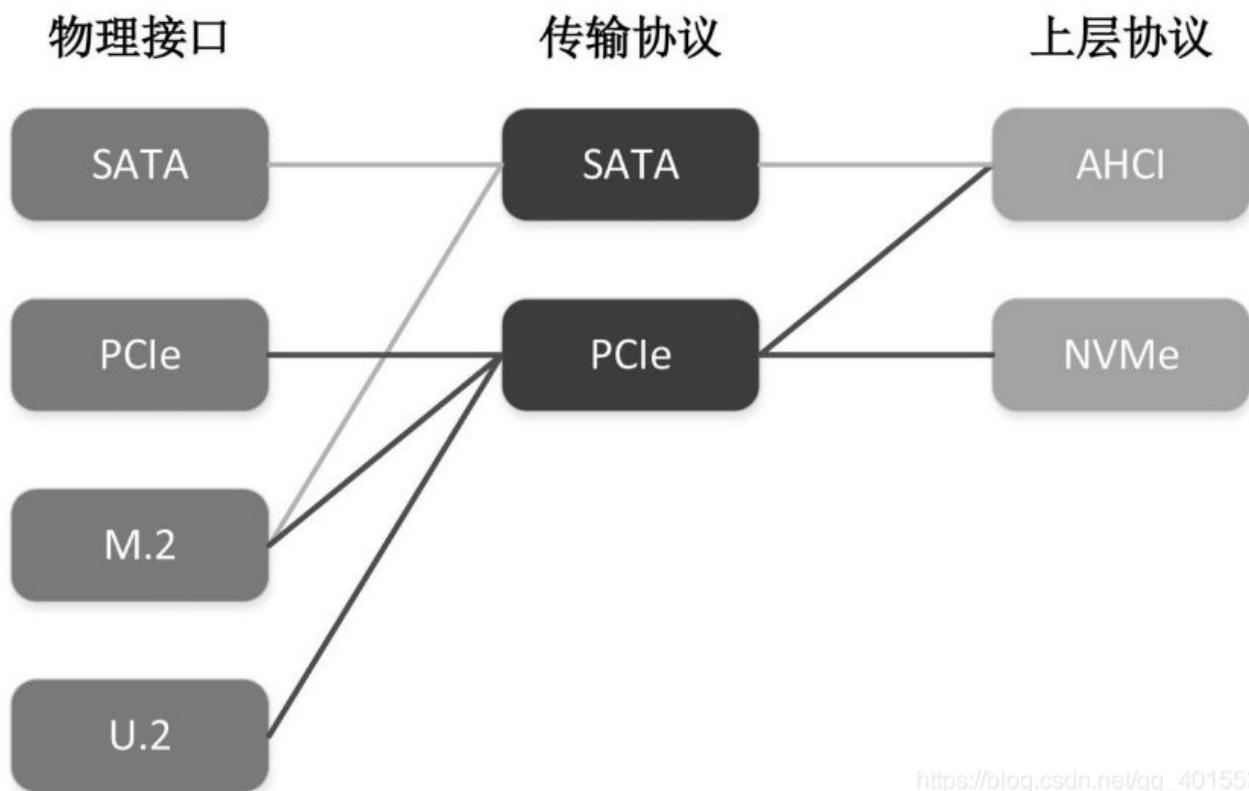
    if !msi_found {
        info!("MSI not found");
        // Use PCI legacy interrupt instead
        // IO Space | MEM Space | Bus Mastering | Special Cycles
        am.write32(ops, loc, PCI_COMMAND, (orig | 0xf) as u32);
    }
}

```

- 第10行中根据命令寄存器的含义写入特定值, 这里主要是关闭传统中断

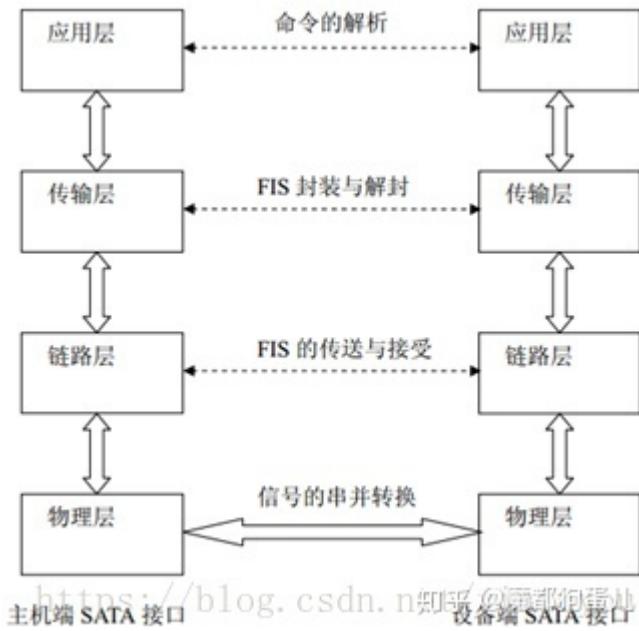
- 第14行读取能力链表的起始位置
- 第18行检查是否支持MSI中断
- 第23行根据loongarch的7a1000桥片设置中断消息目标地址
- 第30/33设置irq
- 第37行启用MSI中断
- 第47行如果发现不支持MSI中断则使用默认中断

SATA协议与AHCI协议



https://blog.csdn.net/qq_40155300

SATA接口协议借鉴TCP/IP模型，将SATA接口划分为四个层次来实现，包括物理层、链路层、传输层、应用层，其体系结构如图所示：



物理层采用全双工串行传输方式，主要功能是进行信号的串并及并串转化。物理层接收来自链路层的数据信息，将接收到的并行的数字逻辑信号转换为串行的差分物理信号，发送到主机端。相应的物理层能将来自主机端的串行差分物理信号转化为并行的逻辑信号传送到链路层。

链路层的主要功能是通过控制原语的传递来控制信息帧的整个传输过程，保证帧信息能够正确的发送与接收并能进行流量的控制，防止数据发送过快或接受过多。

传输层主要负责FIS帧信息结构的封装与解封。1) 传输层接收到来自应用层的数据传输操作请求后，将相关寄存器中信息按SATA协议规定的标准格式封装为FIS传递给链路层。当链路层正确接收完成后，能给传输层反馈成功完成本次传输的信号。2) 传输层接收到来自链路层的SOF信号后，能接收FIS信息帧，并能判断该FIS的类型，根据FIS类型，判断该FIS是否是有效的FIS。如果是则将该FIS中的命令和数据等按照SATA协议规定进行解析，映射到各个寄存器中，然后能通知应用层接收相应寄存器的值。如果该FIS无效，则丢弃。

应用层能够进行接受来自主机端的命令，根据命令的要求将自身的信息发送给主机端，或是接收来自主机端的以PIO或DMA方式传输的数据，同时写入闪存中，也能从闪存中以PIO或DMA的方式读出数据，传送给主机端。在应用层采用两个FIFO对数据进行缓冲，一个为读FIFO，一个为写FIFO。应用层能接收来自传输层的数据帧送入写FIFO中或将来自总线的数据保存在读FIFO中，然后通知传输层构造数据帧。

AHCI (高级主机控制器接口) 由 Intel 开发，以方便处理 SATA 设备。AHCI 规范强调 AHCI 控制器 (称为主机总线适配器，或 HBA) 被设计为系统内存和 SATA 设备之间的数据移动引擎。它封装了 SATA 设备，并为主机提供了标准的 PCI 接口。系统设计人员可以使用系统内存和内存映射寄存器轻松访问 SATA 驱动器，而无需像 IDE 那样处理烦人的任务文件。AHCI 控制器最多可支持 32 个端口，这些端口可以连接不同的 SATA 设备，例如磁盘驱动器、端口倍增器或机箱管理桥。AHCI 支持所有原生 SATA 功能，例如命令队列、热插拔、电源管理等。对于软件开发人员而言，AHCI 控制器只是具有总线主控功能的 PCI 设备。

简单来说，AHCI的抽象层次更高，作为开发者来说可以直接操作AHCI完成硬盘的读写。

主机通过系统内存和内存映射寄存器与 AHCI 控制器通信。最后一个PCI基地址寄存器(BAR[5], header offset 0x24)指向AHCI基内存，称为ABAR(AHCI Base Memory Register)。所有 AHCI 寄存器和存储器都可以通过 ABAR 定位。在上面代码中我们也获取了BAR[5]的基地址和大小并根据此构建AHCI,这里我们不深入讲解AHCI协议的内容，文末会给出链接给想了解的同学。

SATA设备使用DMA的方式进行硬盘的读取，因此需要我们在内存中分配相应的区域，这需要我们为其实现相应的接口：

```
impl provider::Provider for Provider {
    const PAGE_SIZE: usize = PAGE_SIZE;
    fn alloc_dma(size: usize) -> (usize, usize) {
        let pages = size / PAGE_SIZE;
        let mut base = 0;
        for i in 0..pages {
            let frame = frame_alloc().unwrap();
            let frame_pa: PhysAddr = frame.ppn.into();
            let frame_pa = frame_pa.into();
            core::mem::forget(frame);
            if i == 0 {
                base = frame_pa;
            }
            assert_eq!(frame_pa, base + i * PAGE_SIZE);
        }
        let base_page = base / PAGE_SIZE;
        info!("virtio_dma_alloc: {:#x} {}", base_page, pages);
        (base, base)
    }

    fn deallocate_dma(va: usize, size: usize) {
        info!("deallocate_dma: {:x} {:x}", va, size);
        let pages = size / PAGE_SIZE;
        let mut pa = va;
        for _ in 0..pages {
            frame_dealloc(PhysAddr::from(pa).into());
            pa += PAGE_SIZE;
        }
    }
}
```

主要就是需要保证在进行分配时需要保证页帧的连续性，并且由于生命周期的原因，alloc_dma函数结束后，由于没有明确的保存申请的物理页帧，因此可能会被回收掉，这里使用第十行使用core::mem::forget(frame)使得编译期忽略掉页帧，即不会对其调用drop进行回收。

参考链接

[SATA](#)

[AHCI](#)

[AHCI](#)

第七章

第七章引入了管道与文件重定向的功能，以及信号机制。在文件系统完成后，这些功能都可以在软件层面实现，不再设计硬件相关，读者可以阅读rcore中内容完成实验。文末给出的链接提供了linux下更多的进程间通信方式，读者可以在实验的基础上添加

参考链接

[linux进程间通信](#)

[内存映射](#)

[共享内存](#)

第八章

第八章中实现了线程的抽象，在rcore官方文档中可能并没有详细介绍如何完成各个部分的拆分和修改，因此需要读者阅读源代码完成相应的实现，同时，由于前文介绍的loongarch特殊机制，可能一些多余的东西需要删除掉，设计的内容主要是内核栈和trap页以及跳板页的差异，在涉及到这些资源分配和释放的地方，需要对照loongarch版本的源代码进行修改。但总体来说是比较任意的。阅读源代码的思路可以参考如下：

1. 线程控制块
2. 进程控制块
3. 线程创建+线程资源获取
4. 进程创建
5. 线程系统调用wait-tid
6. 进程系统调用wait-pid
7. 线程退出
8. 进程退出

线程

为了支持多道程序并发执行，操作系统引入了进程的概念。进程是程序在特定数据集合上的执行实例，一般由程序、数据集合和进程控制块三部分组成。进程控制块包括很多信息，它记录每个进程运行过程中虚拟内存地址、打开文件、锁和信号等资源的情况。操作系统通过分时复用、虚拟内存等技术让每个进程都觉得自己拥有一个独立的 CPU 和独立的内存地址空间。切换进程时需要切换进程上下文。

线程是程序代码的一个执行路径。一个进程可以包含多个线程，这些线程之间共享内存空间和打开文件等资源，但逻辑上拥有独立的寄存器状态和栈。线程可以由操作系统内核管理，也可以由用户态的线程库管理，或者两者混合。本小节在第四章的基础上将进程进行细分形成进程控制块和线程控制块。

首先我们将线程共享的地址空间和文件资源统一在一起形成进程控制块

```
// 进程控制块
pub struct ProcessControlBlock {
    // immutable
    pub pid: PidHandle,
    // mutable
    inner: UPSafeCell<ProcessControlBlockInner>,
}

pub struct ProcessControlBlockInner {
    pub is_zombie: bool,
    pub memory_set: MemorySet, // 地址空间
    pub parent: Option<Weak<ProcessControlBlock>>, // 父进程
    pub children: Vec<Arc<ProcessControlBlock>>, // 子进程
    pub exit_code: i32,
    pub fd_table: Vec<Option<Arc<dyn File + Send + Sync>>>, // 文件描述符表
    pub signals: SignalFlags, // 信号
    pub tasks: Vec<Option<Arc<TaskControlBlock>>>, // 线程控制块
    pub task_res_allocator: RecycleAllocator, // 资源分配器
    pub mutex_list: Vec<Option<Arc<dyn Mutex>>>, // 互斥锁列表
    pub semaphore_list: Vec<Option<Arc<Semaphore>>>, // 信号量列表
    pub condvar_list: Vec<Option<Arc<Condvar>>>, // 条件变量列表
}
```

将每个线程自己的资源组合在一起形成线程控制块

```

pub struct TaskControlBlock {
    // immutable
    pub process: Weak<ProcessControlBlock>, //所属进程
    // mutable
    inner: UPSafeCell<TaskControlBlockInner>,
}
pub struct TaskControlBlockInner {
    pub kstack: KernelStack,           //每个线程都存在内核栈，其trap上下文位于内核栈上
    pub res: Option<TaskUserRes>,    //线程资源
    pub task_cx: TaskContext,         //线程上下文
    pub task_status: TaskStatus,      //线程状态
    pub exit_code: Option<i32>,       //线程退出码
}

```

在申请线程资源的实现中，根据之前的说明，我们将不会为应用程序构建trap页和跳板页，线程的内核栈也会直接从页帧管理器中分配

```

/// 申请线程资源
pub fn alloc_user_res(&self) {
    let process = self.process.upgrade().unwrap();
    let mut process_inner = process.inner_exclusive_access();
    // alloc user stack
    let ustack_bottom = ustack_bottom_from_tid(self.ustack_base, self.tid);
    let ustack_top = ustack_bottom + USER_STACK_SIZE;
    process_inner.memory_set.insert_area(
        ustack_bottom.into(),
        ustack_top.into(),
        MapPermission::default() | MapPermission::W,
    );
}

```

线程资源分配中主要是为线程分配用户栈。在这里需要注意在构建地址空间中需要删除掉分配用户栈的代码。

在线程创建或者进程创建的函数中，只需要注意将trap上下文保存在内核栈中，就像前面章节的一样。

在线程调度中，除了一如既往的修改根页表基址和修改ASID外，此时我们还增加了无效页表项的指令

```

pub fn run_tasks() {
    loop {
        let mut processor = PROCESSOR.exclusive_access();
        if let Some(task) = fetch_task() {
            let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
            // access coming task TCB exclusively
            let pid = task.process.upgrade().unwrap().getpid(); //应用进程号
            let pgd = task.get_user_token() << PAGE_SIZE_BITS;
            Pgdl::read().set_val(pgd).write(); //设置根页表基地址
            Asid::read().set_asid(pid as u32).write(); //设置ASID
            let mut task_inner = task.inner_exclusive_access();
            let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
            task_inner.task_status = TaskStatus::Running;
            // 在进行线程切换的时候
            // 地址空间是相同的，并且pgd也是相同的
            // 每个线程都有自己的内核栈和用户栈，用户栈互相隔离
            unsafe {
                asm!("invlrb 0x4,{},$r0", in(reg) pid);
            }
            drop(task_inner);
            // release coming task TCB manually
            processor.current = Some(task);
            // release processor manually
            drop(processor);
            unsafe {
                __switch(idle_task_cx_ptr, next_task_cx_ptr);
            }
        }
    }
}

```

当一个进程创建多个线程时，可能发生的情况是一个线程已经退出了，但它的地址映射关系仍然存在TLB中，此时如果新建一个线程的话，新线程将会使用旧线程的空间，比如典型的用户栈，那么在新线程开始运行后，由于TLB中存在映射关系，那么新线程将会使用这个错误关系进行地址转换，从而造成访问错误。这里的解决方法比较粗暴，会直接将进程对应地址空间无效掉，等于是每次都需要重新建立映射关系，但TLB重填的指令较短，我们可以忽略其带来的性能损失。

其它

本章节主要是一些额外的内容较少，目前只写了loongarch中断与内核堆栈回溯工具。后续可能会添加其它内容

中断系统

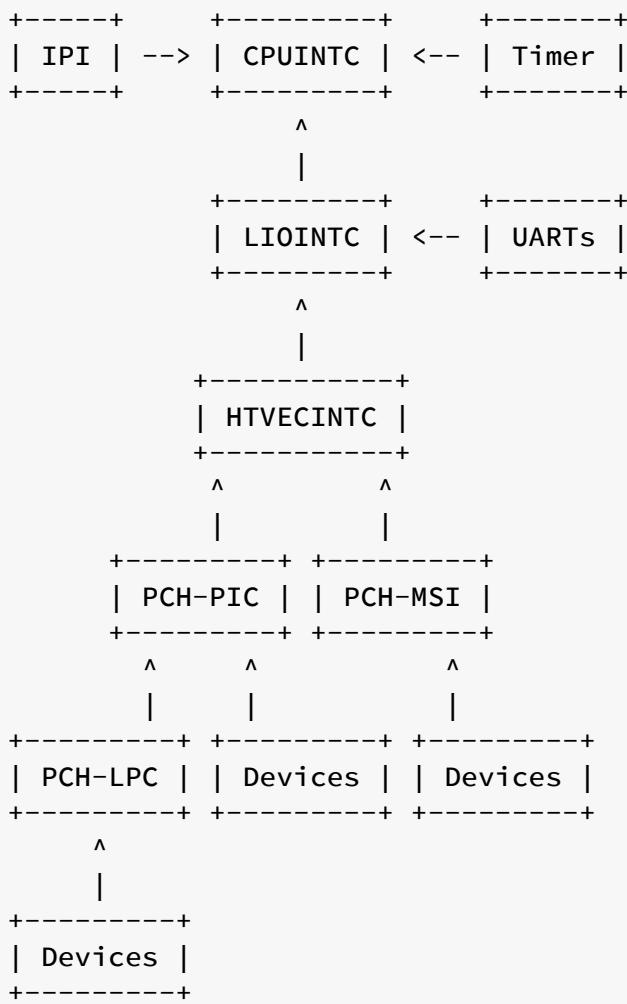
LoongArch的IRQ芯片模型（层级关系）

目前，基于LoongArch的处理器（如龙芯3A5000）只能与LS7A芯片组配合工作。LoongArch计算机中的中断控制器（即IRQ芯片）包括CPUINTC（CPU Core Interrupt Controller）、LIOINTC（Legacy I/O Interrupt Controller）、EIOINTC（Extended I/O Interrupt Controller）、HTVECINTC（Hyper-Transport Vector Interrupt Controller）、PCH-PIC（LS7A芯片组的主中断控制器）、PCH-LPC（LS7A芯片组的LPC中断控制器）和PCH-MSI（MSI中断控制器）。

CPUINTC是一种CPU内部的每个核本地的中断控制器，LIOINTC/EIOINTC/HTVECINTC是CPU内部的全局中断控制器（每个芯片一个，所有核共享），而PCH-PIC/PCH-LPC/PCH-MSI是CPU外部的中断控制器（在配套芯片组里面）。这些中断控制器（或者说IRQ芯片）以一种层次树的组织形式级联在一起，一共有两种层级关系模型（传统IRQ模型和扩展IRQ模型）。

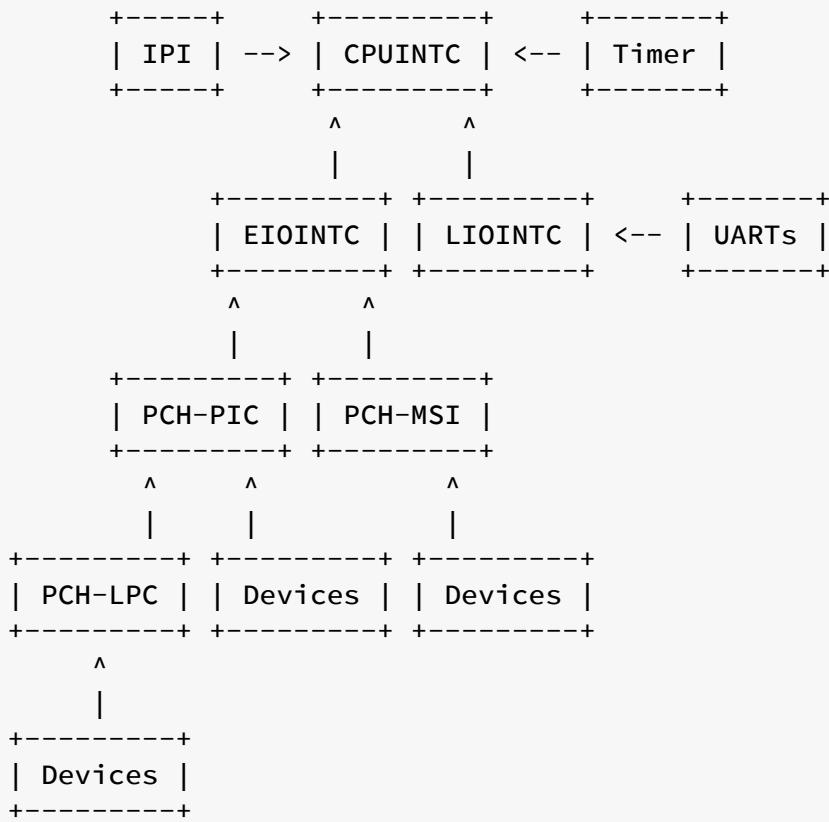
2.1. 传统IRQ模型

在这种模型里面，IPI（Inter-Processor Interrupt）和CPU本地时钟中断直接发送到CPUINTC，CPU串口（UARTs）中断发送到LIOINTC，而其他所有设备的中断则分别发送到所连接的PCH-PIC/PCH-LPC/PCH-MSI，然后被HTVECINTC统一收集，再发送到LIOINTC，最后到达CPUINTC：



2.2. 扩展IRQ模型

在这种模型里面，IPI (Inter-Processor Interrupt) 和CPU本地时钟中断直接发送到CPUINTC，CPU串口 (UARTs) 中断发送到LIOINTC，而其他所有设备的中断则分别发送到所连接的PCH-PIC/PCH-LPC/PCH-MSI，然后被EIOINTC统一收集，再直接到达CPUINTC：



- CPUINTC：即《龙芯架构参考手册卷一》第7.4节所描述的CSR.ECFG/CSR.ESTAT寄存器及其中断控制逻辑，也即是前文所述的13个中断
- LIOINTC：即《龙芯3A5000处理器使用手册》第11.1节所描述的“传统I/O中断”；
- EIOINTC：即《龙芯3A5000处理器使用手册》第11.2节所描述的“扩展I/O中断”；
- HTVECINTC：即《龙芯3A5000处理器使用手册》第14.3节所描述的“HyperTransport中断”；
- PCH-PIC/PCH-MSI：即《龙芯7A1000桥片用户手册》第5章所描述的“中断控制器”；
- PCH-LPC：即《龙芯7A1000桥片用户手册》第24.3节所描述的“LPC中断”。

根据扩展I/O中断，本实验参考张老师给出的代码也实现了键盘、鼠标的中断。

基本步骤是：

1. 初始化扩展I/O中断
2. 初始化7A1000桥片的中断控制器
3. 开启13个中断的对应位
4. 开启全局中断

这里扩展I/O中断实现如下：

```

/// 初始化外部中断
pub fn extioi_init() {
    let mut enable = 0;
    enable
        .set_bit(KEYBOARD_IRQ, true)
        .set_bit(MOUSE_IRQ, true)
        .set_bit(UART0_IRQ, true);
    info!("extioi_init: enable = {:#b}", enable);
    // 使能外部设备中断
    iocsr_write_d(LOONGARCH_IOCSR_EXTIOI_EN_BASE, enable);
    // extioi[31:0] map to cpu irq pin INT1, other to INT0
    //路由到INT1上
    iocsr_write_b(LOONGARCH_IOCSR_EXTIOI_MAP_BASE, 0x1);
    // extioi IRQ 0-7 route to core 0, use node type 0
    //路由到EXT_IOC_NODE_TYPE0指向的0号处理器上
    iocsr_write_w(LOONGARCH_IOCSR_EXTIOI_ROUTE_BASE, 0x0);
    // nodetype0 set to 1, always trigger at node 0 */
    //固定分发模式时,只在0号处理器上触发
    iocsr_write_h(LOONGARCH_IOCSR_EXRIOI_NODETYPE_BASE, 0x1);

    //检查扩展i/o触发器是不是全0, 即没有被触发的中断
    let extioi_isr = iocsr_read_b(LOONGARCH_IOCSR_EXTIOI_ISR_BASE);
    info!("extioi_init: extioi_isr = {:#b}", extioi_isr);
    let current_trigger = extioi_claim();
    info!("extioi_init: current_trigger = {:#b}", current_trigger);
    assert_eq!(extioi_isr, 0);
}

```

桥片中断初始化如下:

```
/// 初始化ls7a中断控制器
pub fn ls7a_intc_init() {
    // enable uart0/keyboard/mouse
    // 使能设备的中断
    ls7a_write_w(
        LS7A_INT_MASK_REG,
        !(0x1 << UART0_IRQ) | (0x1 << KEYBOARD_IRQ) | (0x1 << MOUSE_IRQ)),
    );
    // 触发方式设置寄存器
    // 0: 电平触发中断
    // 1: 边沿触发中断
    // 这里设置为电平触发
    ls7a_write_w(
        LS7A_INT_EDGE_REG,
        0x1 << (UART0_IRQ | KEYBOARD_IRQ | MOUSE_IRQ),
    );
    // route to the same irq in extioi, pch_irq == extioi_irq
    ls7a_write_b(LS7A_INT_HIMSI_VEC_REG + UART0_IRQ, UART0_IRQ as u8);
    ls7a_write_b(LS7A_INT_HIMSI_VEC_REG + KEYBOARD_IRQ, KEYBOARD_IRQ as u8);
    ls7a_write_b(LS7A_INT_HIMSI_VEC_REG + MOUSE_IRQ, MOUSE_IRQ as u8);
    // 设置中断电平触发极性
    // 对于电平触发类型:
    // 0: 高电平触发;
    // 1: 低电平触发
    // 这里是高电平触发
    ls7a_write_w(LS7A_INT_POL_REG, 0x0);
}
```

代码中出现的常量源代码中也给出了注释。

内核栈回溯工具

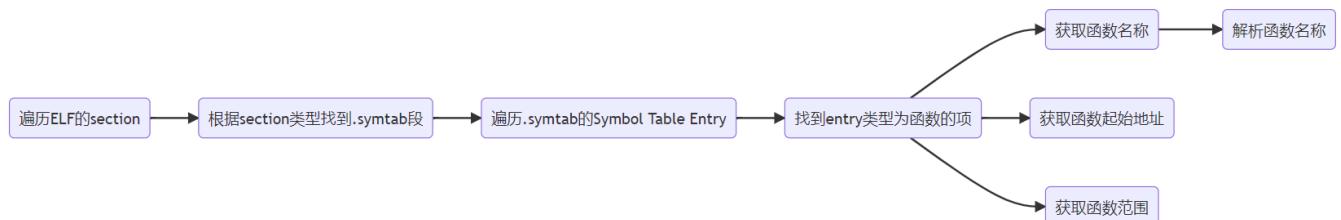
本实验包含了risc-v版本的堆栈回溯工具以及loongarch版本，两者的实现原理相同，这里给出的是risc-v版本说明。

使用说明

此工具用于在 `rCore` panic时进行堆栈回溯，打印函数调用路径。目前只在ch9分支上进行测试，从ch6开启文件系统后均可以使用此工具。

如何获取函数信息

在栈回溯时，需要查询函数信息，而这些函数信息主要包含于可执行文件中。具体的细节可以在 `elf`文件函数信息这里查看，这里给出主要的查找过程



由于 `rust` 会对函数名称进行重整，类似于c++，因此需要使用相应工具进行解析才能转为可读的名称。同时，汇编文件中的函数可能不会被上述过程收集到。

栈回溯分类

主要有两种堆栈回溯方式，一种是使用 `sp` 和 `fp` 指针进行回溯，如下图所示，在这种方式下，每当函数进行开辟栈帧操作后，就会保存 `ra` `tp` 的值，然后令 `fp` 指向当前的栈顶，



在这种情况下，在进行栈回溯时，首先根据 `fp` 寄存器指向的地址，取出保存在函数栈中 `ra` 和 `fp` 寄存器的数据，`ra` 的值是函数返回地址，`fp` 的值是上一级函数栈的栈顶地址，根据 `ra` 的值到收集的函数信息中查找此地址是否位于某个函数的范围，如果是，则记录函数信息，然后根据 `fp` 回到上一级函数，继续读取 `ra` 和 `fp` 的值，指导无法找到对应的函数区间。

第二种回溯方式是由于某些编译器不会利用 `fp` 生成上述的代码，从而需要额外的手段进行解析，有的会使用ELF文件中的 `.eh_frame` 段内容，由于这些方式比较复杂，因此本工具暂不使用此方法。

本工具回溯方法

本工具根据一般函数生成形式，比如rust生成的一段 `risc-v` 代码如下

```
0000000080210412 <my_trace>:
 80210412: 7149          addi    sp,sp,-368
 80210414: f686          sd      ra,360(sp)
 80210416: f2a2          sd      s0,352(sp)
 80210418: eea6          sd      s1,344(sp)
 8021041a: eaca          sd      s2,336(sp)
 8021041c: e6ce          sd      s3,328(sp)
 8021041e: e2d2          sd      s4,320(sp)
 80210420: fe56          sd      s5,312(sp)
 80210422: fa5a          sd      s6,304(sp)
```

可以看到，函数的前两条指令是开辟栈空间和保存 `ra` 的指令，因此这里一个简单的想法就是通过读取函数的第一条指令和第二条指令，获取到开辟的栈空间大小以及 `ra` 存储的位置，这里 `ra` 一般就是存储在栈顶，读取第二条指令主要是确保这条指令是保存 `ra` 的指令。再使用汇编指令读取当前的 `sp` 值，就可以得到下面的回溯方式：

读取函数第一条指令和第二条指令获得栈大小size

栈底: sp

栈顶: sp + size

ra : m[sp+size-8]

寻找ra所在函数

将找到的函数设置为当前函数

再次重复上述过程

因此主要工作在于如何解析函数的第一条和第二条指令，通过查询risc-v手册可以找到各条指令的格式，比如 addi 指令的格式

addi rd, rs1, immediate

$x[rd] = x[rs1] + \text{sext(immediate)}$

加立即数(*Add Immediate*). I-type, RV32I and RV64I.

把符号位扩展的立即数加到寄存器 $x[rs1]$ 上，结果写入 $x[rd]$ 。忽略算术溢出。

压缩形式: c.li rd, imm; c.addi rd, imm; c.addi16sp imm; c.addi4spn rd, imm

31	20 19	15 14	12 11	7 6	0
	immediate[11:0]	rs1	000	rd	0010011

读取第一条指令并按照上面的格式解析出立即数部分就可以得到栈大小，但由于risc-v的编译器会做某些优化，将 addi 指令使用压缩指令表示，而压缩指令一般是两字节格式，比如 c.addi 指令的格式如下：

c.addi rd, imm

$x[rd] = x[rd] + \text{sext(imm)}$

加立即数 (*Add Immediate*). RV32IC and RV64IC.

扩展形式为 addi rd, rd, imm.

15	13	12	11	7 6	2 1	0
000	imm[5]		rd	imm[4:0]	01	

因此需要根据压缩指令和未压缩的指令共同判断第一条指令是否未开辟栈空间的指令和栈空间大小。同理，判断第二条指令也需要如上的工作。

具体的实现请查看源代码。

使用方法

本工具以一个库的形式提供，需要传入的参数为OS的可执行文件。

```

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    if let Some(location) = info.location() {
        println!(
            "[kernel] Panicked at {}:{} {}",
            location.file(),
            location.line(),
            info.message().unwrap()
        );
    } else {
        println!("[kernel] Panicked: {}", info.message().unwrap());
    }
    unsafe {
        backtrace();
    }
    shutdown(255)
}

use stack_trace::{Trace};
unsafe fn backtrace() {
    let mut os_name:Vec<&str> = Vec::new();
    let all_file = ROOT_INODE.ls();
    all_file.iter().for_each(|x| {
        if x.contains("os") {
            os_name.push(x);
        }
    });
    os_name.sort(); //由于内核文件较大，会被分成多个文件
    let mut data = Vec::new();
    os_name.iter().for_each(|name| {
        let mut file = open_file(*name,OpenFlags::RDONLY).unwrap();
        let d = file.read_all();
        trace!("name: {} {}", name, d.len());
        data.extend_from_slice(d.as_slice());
    });
    let mut trace = Trace::new();
    trace.init(data.as_slice()); //初始化，传入内核可执行文件
    let road = trace.trace(); //收集函数调用信息
    road.iter().for_each(|s|{
        println!("{} {}", s);
    });
}

```

目前的内核无法传递参数，为了在内核中读取本身的ELF文件，需要在编译后将ELF文件与应用程序文件一样打包在fs.img中，为了不将代码写死，这里在easy-fs-fuse添加了一个参数：

```

.arg(
    Arg::with_name("kernel")
        .short("k")
        .long("kernel")
        .takes_value(true)
        .help("Kernel source dir(with backslash)"),
)

```

同时在os的Makefile文件也需要修改相应的参数：

```
@cd .. /easy-fs-fuse && cargo run --release -- -s .. /user/src/bin/ -t  
.. /user/target/riscv64gc-unknown-none-elf/release/ -k .. /os/target/riscv64gc-  
unknown-none-elf/release/
```

然后只需要在easy-fs-fuse将内核文件写入fs-img, 由于内核可执行文件高达16MB, 并且os中文件系统最多支持8MB大小的文件, 因此在打包时需要将文件进行切分, 并且设置fs-img大小为32MB

```
let block_file : Arc<BlockFile> = Arc::new( data: BlockFile(Mutex::new( t {  
    let f : File = OpenOptions::new()  
        .read(true) : &mut OpenOptions  
        .write(true) : &mut OpenOptions  
        .create(true) : &mut OpenOptions  
        .open( path: format!("{}{}", target_path, "fs.img"))?  
    f.set_len(32 * 2048 * 512).unwrap();  
    f  
})));  
// 16*2MiB, at most 4095*2 files  
let efs : Arc<Mutex<EasyFileSystem>> = EasyFileSystem::create( block_device: block_file, total_blocks: 32 * 2048, inode_bitmap_blocks: 1);  
  
let kernel_path = matches.value_of("kernel").unwrap();  
println!("kernel path = {}{}", kernel_path, "os");  
let mut file = File::open(format!("{}{}", kernel_path, "os")).unwrap();  
let mut all_data: Vec<u8> = Vec::new();  
file.read_to_end(&mut all_data).unwrap();  
println!("{}", all_data.len());  
//如果数据大于8MB,将数据切分  
let mut data_vec: Vec<Vec<u8>> = Vec::new();  
let mut data_vec_len = 0;  
while data_vec_len < all_data.len() {  
    let mut data_vec_tmp: Vec<u8> = Vec::new();  
    data_vec_tmp.extend_from_slice(&all_data[data_vec_len..(data_vec_len +  
8 * 1024 * 1024).min(all_data.len())]);  
    data_vec.push(data_vec_tmp);  
    data_vec_len += 8 * 1024 * 1024;  
}  
for i in 0..data_vec.len() {  
    let inode = root_inode.create(format!("os{}", i).as_str()).unwrap();  
    inode.write_at(0, data_vec[i].as_slice());  
}
```

完成上述步骤, 就可以在内核中读取本身的ELF文件并传入栈回溯库了(内核读取这种大文件有点慢)

目前在内核中使用此功能可以达到的效果如下所示

```
initproc
infloop
sync_sem
pipe_large_test
filetest_simple
fantastic_text
os0
os1
*****
[kernel] Panicked at src/test.rs:14 test panic
0x80210438 (+16) stack_trace::trace::Trace::trace
0x80206db8 (+1116) os::lang_items::backtrace
0x80206c1e (+196) rust_begin_unwind
0x8022243c (+44) core::panicking::panic_fmt
0x80205880 (+104) os::test::test_stack_trace
0x8020ae1e (+428) rust_main
make: *** [Makefile:109: run-inner] 错误 1
```

```
os0
os1
*****
[kernel] Panicked at src/task/process.rs:144 [KERNEL_EXEC]
0x8021dc5a (+16) stack_trace::trace::Trace::trace
0x80209f52 (+1116) os::lang_items::backtrace
0x80209db8 (+196) rust_begin_unwind
0x802301e2 (+44) core::panicking::panic_fmt
0x8020e56a (+320) os::task::process::ProcessControlBlock::exec
0x8021b4ae (+510) os::syscall::process::sys_exec
0x80217c50 (+168) trap_handler
make: *** [Makefile:109: run-inner] 错误 1
```

注意事项

目前的实现仍然比较简陋，且限制较大，由于在出错时需要读取文件内容，此时如果发生任务调度可能会造成死锁的问题，但这是在ch9会发生的现象，如果在前面的章节中，读取文件内容是阻塞式的不会发生任务调度，因此应该不会造成死锁问题。

为了解决这个问题，可以在开始进入用户态之前就读取内核数据，后面如果发生错误，就不需要读取文件发生死锁。上面的代码修改为：

```

lazy_static!{
    static ref KERNEL_DATA: UPIIntrFreeCell<Vec<u8>> =
unsafe{UPIIntrFreeCell::new(Vec::new())};
}

pub fn init_kernel_data(){
    let mut os_name:Vec<&str> = Vec::new();
    let all_file = ROOT_INODE.ls();
    all_file.iter().for_each(|x| {
        if x.contains("os") {
            os_name.push(x);
        }
    });
    os_name.sort();
    os_name.iter().for_each(|name| {
        let mut file = open_file(*name,OpenFlags::RDONLY).unwrap();
        let d = file.read_all();
        trace!("name: {} {}",name,d.len());
        KERNEL_DATA.exclusive_access().extend_from_slice(d.as_slice());
    });
}

unsafe fn backtrace() {
    let mut trace = Trace::new();
    trace.init(KERNEL_DATA.exclusive_access().as_slice());
    let road = trace.trace();
    road.iter().for_each(|s|{
        println!("{}",s);
    });
}

```

在main函数中，在开始进入用户程序之前调用 `init_kernel_data()` 即可。

改进

- 在编译前获取函数信息并与内核一同链接 --> 适用于所有章节的栈回溯方法
- 使用 `.eh_frame` 进行栈回溯而不是读取函数的前两条指令

VBE图形显示

待完成