**面向LoongArch32指令集的uCore操作系统调试分析与优化技术文档**

小组名称：龙舌兰日落  
小组成员: 吴睿喆  
指导老师：陈咸彰

学校：重庆大学

# 1 项目介绍

uCore操作系统是清华大学在MIT的xv6（基于MIT License）操作系统基础上开发的一个教学操作系统。重庆大学陈咸彰老师团队在uCore的基础上，面向LoongArch32指令集设计开发了一套操作系统实验，已经以实验课程的形式发布在头歌平台上（请见参考资料1）。实验的内容主要是对面向LoongArch32指令集的uCore操作系统进行调试分析，进而做一些小的改写或优化。LoongArch32是龙芯自主研发的32位指令集架构，具有完全自主知识产权。将uCore移植到LoongArch32平台上，不仅有助于验证该指令集的功能和性能，还能为教学和科研提供宝贵的实践平台。

调试技术：利用LoongArch32的调试工具和方法，对uCore进行有效的调试，定位和解决潜在问题。

性能分析：使用性能分析工具，评估uCore在LoongArch32平台上的运行效率，识别性能瓶颈。

优化策略：针对LoongArch32架构的uCore优化方法，提高系统的整体性能和稳定性。

在本次比赛中，需要完成七部分的实训内容，分别为：异常与中断、内存管理、进程管理、同步与互斥、文件系统、I/O管理、磁盘调度。

# 2 uCore操作系统调试分析与优化

**2.1 异常与中断**

异常（Exception）和中断（Interrupt）是操作系统处理中断机制的核心概念。在LoongArch32架构中，中断被视为异常的一种类型。异常包括多种类型，

例如：中断（EX\_IRQ）：由外部设备引发的中断信号。TLB异常：包括加载操作页无效（EX\_TLBL）、存储操作页无效（EX\_TLBS）和TLB重填（EX\_TLBR）。非法指令（EX\_RI）：执行了不存在的指令。特权等级错误（EX\_IPE）：指令的特权等级不符合要求。系统调用（EX\_SYS）：用户程序请求操作系统服务。地址错误（EX\_ADE）：例如，地址未对齐导致的错误。

处理器在异常发生时，会根据异常类型跳转到相应的异常处理入口，保存当前的执行状态，并执行预设的异常处理程序。在LoongArch32架构中，处理器提供了两个异常入口：常规异常入口和TLB异常入口。常规异常入口的地址存储在CSR寄存器的EBASE字段中，TLB异常入口的地址存储在CSR寄存器的RFBASE字段中。操作系统通过设置这些入口地址，确保在异常发生时能够正确跳转到相应的处理程序。

时钟中断是由处理器的定时器产生的，用于实现操作系统的时间管理和任务调度。在LoongArch32架构中，定时器相关的控制状态寄存器（CSR）用于配置和管理定时器。操作系统可以通过设置定时器，使其在特定时间间隔产生中断，从而实现周期性的任务调度。

串口中断则与串行通信接口（如NS16550a兼容的串口控制器）相关。当串口接收到输入数据时，会产生中断，通知操作系统进行数据读取和处理。在LoongArch32的QEMU模拟环境和Chiplab SoC中，串口中断通常连接到硬件中断线HWI[0]。

通过对异常和中断的有效管理，操作系统能够及时响应外部设备的请求，处理异常情况，确保系统的稳定运行。

首先实现时钟中断处理函数clock\_int\_handler()，修改完整代码如下:

int clock\_int\_handler(void \* data)

{

#ifdef LAB1\_EX2

*// LAB1 EXERCISE2: YOUR CODE*

*// (1) count ticks here*

  ticks ++;

#ifdef \_SHOW\_100\_TICKS

*// (2) if ticks % 100 == 0 then call kprintf to print "100 ticks"*

  if (ticks % 100 == 0) {

    kprintf("100 ticks\n");}

#endif

#endif

#ifdef LAB4\_EX1

  run\_timer\_list();

#endif

  reload\_timer();

  return 0;

}

接着实现串口中断处理函数serial\_int\_handler():

void serial\_int\_handler(void \*opaque)

{

    unsigned char id = inb(COM1+COM\_IIR);

    if(id & 0x01)

        return ;

*//int c = serial\_proc\_data();*

    int c = cons\_getc();

#if defined(LAB1\_EX3) && defined(\_SHOW\_SERIAL\_INPUT)

*// LAB1 EXERCISE3: YOUR CODE*

    kprintf("got input %c\n",c);

#endif

#ifdef LAB4\_EX2

    extern void dev\_stdin\_write(char c);

    dev\_stdin\_write(c);

#endif

}

在以上两个函数实现后控制台随时间不断输出‘100ticks’，当用户输入随意字符，产生中断，中断正在打印的“100ticks”并输出“got input”+该字符，然后将处理器再交由原先进程。

**2.2 内存管理**

实验主要实现uCore内核对物理内存的管理，通过kern\_init函数初始化并调用pmm\_init完成物理内存的初始化、分配以及虚拟内存的映射。在LoongArch32版本中，由于简化设计，物理内存大小固定为32MB，并通过划分固定大小页面（4KB）作为最小分配单位实现内存管理。实验中，内核需构建空闲页面链表，完成二级页表建立并启用分页机制，从而支持虚拟地址到物理地址的映射及缺页异常处理。关键实现包括物理页的管理数据结构Page及物理内存页管理器pmm\_manager框架，核心函数如boot\_map\_segment和get\_pte用于实现页表构建和虚实地址映射。通过该实验，可以掌握物理内存管理和虚拟化的核心机制，深入理解操作系统的内存分配原理和实现方法。

首先实现first-fit连续物理内存分配算法，first-fit分配算法需要维护一个查找有序（地址按从小到大排列）空闲块（以页为最小单位的连续地址空间）的数据结构，本实验中使用了libs/list.h定义的可挂接任意元素的通用双向链表，可以完成对双向链表的初始化/插入/删除等。但是在不改变first fit算法的情况下，default\_free\_pages函数中存在可优化空间。在default\_free\_pages函数中，是通过遍历整个链表寻找可以合并的前驱后继节点的，时间复杂度是O(N)。如果把链表数据结构换成树结构等可以通过内存地址进行快速检索的结构，就可以降低default\_free\_pages函数的时间复杂度。以下是对算法的优化：

default\_alloc\_pages(size\_t n) {

#ifdef LAB2\_EX1

    assert(n > 0);

    if (n > nr\_free) {

        return NULL;

    }

    struct Page \*page = NULL;

    list\_entry\_t \*le = &free\_list;

*// TODO: optimize (next-fit)*

    int THRESHOLD = 1;

    if(page->property >= n){

        if(page->property - n < THRESHOLD){

            nr\_free -= page->property;

        }

        else{

            struct Page \*new\_page = page + n;

            nr\_free -= n;

            new\_page->property = page->property - n;

            SetPageProperty(new\_page);

            list\_add\_after(&(page->page\_link), &(new\_page->page\_link));

        }

        list\_del(&(page->page\_link));

        ClearPageProperty(page);

}

    while ((le = list\_next(le)) != &free\_list) {

        struct Page \*p = le2page(le, page\_link);

        if (p->property >= n) {

            page = p;

            break;

        }

    }

    if (page != NULL) {

        if (page->property > n) {

            struct Page \*p = page + n;

            p->property = page->property - n;

            SetPageProperty(p);

            list\_add\_after(&(page->page\_link), &(p->page\_link));

        }

        list\_del(&(page->page\_link));

        nr\_free -= n;

        ClearPageProperty(page);

    }

    return page;

#endif

}

优化的方向主要是first-fit算法最后在多次分配过后会出现很多小的外部碎片，这些小碎片已经很小，很难被再次利用，所以可以设定一个阈值，在内存页的数量小于这一阈值时将这一块连续空闲块的页面全部分配出去，有利于减少空闲链表的长度，加快查找的效率，同时在分配时也不需要划分空闲页，加快了分配的速度。

接着实现寻找虚拟地址对应的页表项：

get\_pte(pde\_t \*pgdir, uintptr\_t la, bool create) {

#ifdef LAB2\_EX2

pde\_t \*pdep = NULL; // 寻找一级页表中的索引，入口

pdep = pgdir + PDX(la); // PDX(la) 前10位(PDE)

// 检查页表项是否存在

if (((\*pdep) & PTE\_P) == 0)

{

// 不需要分配或者分配的页为NULL

if (!create)

return NULL;

// 设置引用次数一次

struct Page \*new\_pte = alloc\_page();

if (!new\_pte)

return NULL;

page\_ref\_inc(new\_pte);

// 得到物理地址

uintptr\_t pa = (uintptr\_t)page2kva(new\_pte);

// 清理虚拟地址

memset((void \*)pa, 0, PGSIZE);

kprintf("@@@ %x\n", pa);

// set page directory entry's permission

\*pdep = PADDR(pa);

(\*pdep) |= (PTE\_U | PTE\_P | PTE\_W);

}

pte\_t \*ret = (pte\_t \*)KADDR((uintptr\_t)((pte\_t \*)(PDE\_ADDR(\*pdep)) + PTX(la)));

kprintf("@@GET\_PTE %x %x %x\n", \*pdep, ret, \*ret);

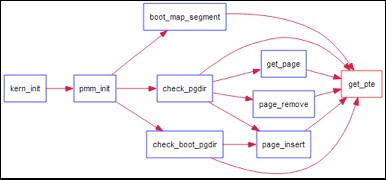
return ret; // return page table entry

#endif

}

代码的思路如下：我们通过一级页表的地址和索引值找到具体的地址，并判断这一页是否存在，如果存在，则直接直接计算出对应的页表项；如果不存在的话，那么检测create参数，他代表着这次如果对应的项不存在的话，是否新建一个页，不需要新建页则直接返回。

如果需要新建一个页就分配空间，如果分配成功则增加页的引用项，算出页的线性地址，再在一级页表中添加页目录项指向表示二级页表的新物理页。并给页表设置相应的权限。



get\_pte函数调用关系

第三步释放虚地址所在页并取消二级页表的映射，在实现上一步的基础上补全page\_remove\_pte()：

page\_remove\_pte(pde\_t \*pgdir, uintptr\_t la, pte\_t \*ptep) {

#ifdef LAB2\_EX3

if (ptep && (\*ptep & PTE\_P))

{

// 找到二级页表entry对应的页面

struct Page \*page = pte2page(\*ptep);

// page 引用减一

page\_ref\_dec(page);

// 如果page引用数为0,则清除页面

if (page\_ref(page) == 0)

{

free\_page(page);

}

// 将二级页表entry清零

\*ptep = 0;

}

// 清洗快表

tlb\_invalidate\_all();

#endif

}

page\_remove\_pte（）函数的主要作用是传入虚地址和二级页表项地址，释放某虚地址所在的物理页并取消对应二级页表项的映射。根据注释内容，设计过程如下：首先通过ptep指针和PTE\_P标志是否都不为零，判断二级页表入口是否存在，若不存在该函数就不需要继续执行了，存在则需要清除对应的页面内容。

接下来，先使用pte2page函数找到ptep指针对应的页面，再通过page\_ref\_dec将页面的引用数减1。之后判断若页面引用数是否为0，如果为0则代表该页面已经空闲，使用free\_page函数释放页面。最后，当页面已经清除，将ptep指针置0，取消对应二级页表项的映射，再使用tlb\_invalidate函数冲刷快表即可。

**2.3 进程管理**

在操作系统实验中，进程管理是关键环节之一。uCore操作系统通过进程控制块（proc\_struct）来管理进程的状态、标识符、内核栈、父进程、内存管理信息等。在内核初始化过程中，首先创建空闲进程（idle），然后创建初始进程（init），进而加载并执行用户态进程。用户进程的虚拟地址空间由页表管理，实现用户态与内核态的隔离。系统调用机制允许用户进程通过陷入内核态来请求操作系统服务。进程的生命周期管理包括进程的创建、执行、等待、退出和资源回收等。在多进程环境中，操作系统通过调度算法（如FIFO）来管理进程的执行顺序和CPU占用时间。此外，进程间的父子关系、进程间通信、进程同步与互斥等也是进程管理的重要内容。

本实验的第一步为创建一个用户进程，此处完成一个优先级的转变，从内核态切换到用户态（即特权级从0到3）。补充代码部分：

#ifdef LAB3\_EX1

/\* LAB3:EXERCISE1 YOUR CODE

\* should set tf\_era,tf\_regs.reg\_r[LOONGARCH\_REG\_SP],tf->tf\_prmd

\* NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE from kernel and enable interrupt. So

\* tf->tf\_prmd should be PLV\_USER | CSR\_CRMD\_IE

\* tf->tf\_era should be the entry point of this binary program (elf->e\_entry)

\* tf->tf\_regs.reg\_r[LOONGARCH\_REG\_SP] should be the top addr of user stack (USTACKTOP)

\*/

tf->tf\_era = elf->e\_entry;

tf->tf\_regs.reg\_r[LOONGARCH\_REG\_SP] = USTACKTOP;

uint32\_t status = 0;

status |= PLV\_USER; // set plv=3(User Mode)

status |= CSR\_CRMD\_IE;

tf->tf\_prmd = status;

#endif

代码中变量tf的含义：tf是一个是中断帧的指针，指向内核栈的某个位置，当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。tf的定义在kern/trap/trap.h中。优先级转变的具体做法如下：1.将tf\_era设置为用户态，这个定义在kern/mm/memlayout.h中，有一个宏定义已经定义了用户态和内核态；2.status也需要设置为用户态；3.需要将tf\_regs.reg\_r[LOONGARCH\_REG\_SP]设置为用户栈的栈顶，直接使用之前建立用户栈时的参数USTACKTOP就可以；4.最后打开中断。

第二步实现父进程创建子进程，通过copy\_range()函数，创建子进程的函数do\_fork()在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。

copy\_range(pde\_t \*to, pde\_t \*from, uintptr\_t start, uintptr\_t end, bool share) {

assert(start % PGSIZE == 0 && end % PGSIZE == 0);

assert(USER\_ACCESS(start, end));

do {

pte\_t \*ptep = get\_pte(from, start, 0), \*nptep;

if (ptep == NULL) {

start = ROUNDDOWN\_2N(start + PTSIZE, PGSHIFT);

continue ;

}

if (\*ptep & PTE\_P) {

if ((nptep = get\_pte(to, start, 1)) == NULL) {

return -E\_NO\_MEM;

}

uint32\_t perm = (\*ptep & PTE\_USER);

struct Page \*page = pte2page(\*ptep);

struct Page \*npage=alloc\_page();

assert(page!=NULL);

assert(npage!=NULL);

int ret=0;

#ifdef LAB3\_EX2

memcpy(page2kva(npage), page2kva(page), PGSIZE); //将父进程的物理页的内容复制到子进程中去

page\_insert(to, npage, start,perm); //建立子进程的物理页与虚拟页的映射关系

#endif

assert(ret == 0);

}

start += PGSIZE;

} while (start != 0 && start < end);

return 0;

}

copy\_range函数实现了Copy-on-Write机制，主要任务是在父进程和子进程之间共享物理页框。首先，通过get\_pte()函数获取父进程对应线性地址的页表项指针，然后判断是否存在，若不存在则继续查找下一页。如果页表项存在且有效，通过get\_pte()函数获取子进程对应线性地址的页表项指针，再通过alloc\_page()函数分配一个新的物理页，然后通过memcpy()函数将父进程物理页的内容复制到子进程的物理页中。最后，通过page\_insert()函数建立子进程的物理页与虚拟页的映射关系，使得父子进程在写入时能够进行独立的复制。

**2.4 同步与互斥**

在操作系统中，同步和互斥是确保并发进程正确执行的关键机制。

互斥：当多个进程需要访问共享资源（如临界区）时，互斥机制确保同一时刻只有一个进程能够访问该资源，其他进程必须等待，直到当前进程释放资源。

同步：用于协调多个进程的执行顺序，确保进程按照预定的顺序执行。例如，一个进程必须等待另一个进程完成特定任务后才能继续执行。

实验实现FIFO管道，半双工通信方式，允许无亲缘关系的进程间通信。在文件系统中以文件形式存在，进程可以通过打开该文件进行读写操作，实现通信。

首先完善PIPE相关数据结构和函数：

int pipealloc(struct pipe\_port \*p0, struct pipe\_port \*p1)

{

pipe\_enty = (struct pipe\*)kmalloc(sizeof(struct pipe));

// LAB5 YOUR CODE EX1 -----------------

// 使用kmalloc为pipe\_port分配空间

if (new\_pipe == NULL) {

return -1; // 分配失败

}

// 使用 kmalloc 为 pipe\_port 分配空间

struct pipe\_port \*port0 = (struct pipe\_port\*) kmalloc(sizeof(struct pipe\_port));

if (port0 == NULL) {

kfree(new\_pipe);

return -1; // 分配失败

}

struct pipe\_port \*port1 = (struct pipe\_port\*) kmalloc(sizeof(struct pipe\_port));

if (port1 == NULL) {

kfree(new\_pipe);

kfree(port0);

return -1; // 分配失败

}

// 分配管道缓冲区

new\_pipe->addr = kmalloc(PIPESIZE);

if (new\_pipe->addr == NULL) {

kfree(new\_pipe);

kfree(port0);

kfree(port1);

return -1; // 分配失败

}

// LAB5 YOUR CODE EX1 -----------------

pipe\_enty->addr= kmalloc(128);

pipe\_enty->readopen = 1;

pipe\_enty->writeopen = 1;

pipe\_enty->nwrite = 0;

pipe\_enty->nread = 0;

ppt0->readable = 1;

ppt0->writable = 0;

ppt1->readable = 0;

ppt1->writable = 1;

ppt0->pipe=pipe\_enty;

ppt1->pipe=pipe\_enty;

return 0;

}

该函数用于对管道进行初始化，并对管道端口分配对应的管道。为两个管道端口分配空间，实现形式如该函数的第一行，两个端口指针为ppt1和ppt0 。

接着实现管道通信并进行编译测试，通过参考管道读取数据函数来实现管道写入数据功能：

int pipewrite(struct pipe \*pi, char\* msg, int len)

{

int w = 0; // 记录已写入的字节数

while (w < len) {

// 检查读端是否关闭

if (pi->readopen == 0) {

return -1;

}

// 管道满时进行阻塞

if (pi->nwrite == pi->nread + PIPESIZE) {

do\_yield();

} else {

// 计算可写入的大小

int size = MIN3(

len - w,

pi->nread + PIPESIZE - pi->nwrite,

PIPESIZE - (pi->nwrite % PIPESIZE)

);

kprintf("size %d\n", size);

// 逐字节写入数据

for (int x = 0; x < size; x++) {

\*(pi->addr + (pi->nwrite + x) % PIPESIZE) = \*(msg + w + x);

}

// 更新写指针和已写入字节数

pi->nwrite += size;

w += size;

}

}

return w;

}

在写函数中，用for循环对大小为size的空间进行写入，写入地址为\*(pi->addr+w+x)（w是已经写入的空间，x是当前for循环遍历到的位置），写入值为\*(msg+x+w)。写入完成后需要将pipe已经写入的统计量nwrit加上size即：pi->nwrite += size;；并将本次写入的大小w加上size，即w += size;

通过上述两步实验，完成对管道通信的编写，在调用该pipe进程中，可以得到输入的字符串，并且输出字符串的size。

**2.5 文件系统**

uCore文件系统仿照UNIX的设计，包含文件、目录项、inode和挂载点等基本概念。系统由多层结构组成：文件访问接口、抽象层、基于索引存储的简单文件系统（Simple FS）以及与硬件交互的设备接口层。通过这些层次，文件系统能够实现文件的读取、写入和创建等操作，inode存储文件元数据，设备驱动程序负责硬件操作。这个实验主要是关于进程管理的实现。需要我们构建一个完整的进程控制系统。其核心是进程控制块(PCB)的设计与实现，它需要包含进程的所有重要信息，如进程状态、ID、父进程指针等。进程可以处于多个状态：未初始化、就绪、运行、睡眠和僵尸状态。我们需要实现进程创建、调度和切换的基本功能。在具体实现中，要设计合适的数据结构来维护进程列表，使用链表来管理所有进程，并实现进程间的切换机制。这些功能将构成操作系统中最基础的进程管理系统，是实现多任务操作系统的关键所在。实验要求我们完整理解和实现进程的生命周期管理，包括进程的创建、运行、调度、挂起和终止等各个环节，还需要考虑实际编程中的各种细节问题，如内存管理、上下文切换等。

首先实现读文件操作：

if((blkoff = offset % SFS\_BLKSIZE)!= 0)//如果文件起始位置不对齐

{

size = (nblks != 0) ? (SFS\_BLKSIZE - blkoff) : (endpos - offset);//计算起始块需要读取的块长度

if((ret = sfs\_bmap\_load\_nolock(sfs, sin, blkno, &ino)) != 0)//计算文件索引号

goto out;

if ((ret = sfs\_buf\_op(sfs, buf, size, ino, blkoff)) != 0)//读文件

goto out;

alen += size;//更新已读取块长度

if (nblks == 0)

goto out;

buf += size, blkno ++, nblks --;//更新缓冲区已读数据规模,当前块,剩余读取块数量

}

size = SFS\_BLKSIZE;//对齐部分单次读取块长度恒定

while(nblks != 0)//循环处理对齐部分

{

if((ret = sfs\_bmap\_load\_nolock(sfs, sin, blkno, &ino)) != 0)//同上

goto out;

if((ret = sfs\_block\_op(sfs, buf, ino, 1)) != 0)

goto out;

alen += size, buf += size, blkno ++, nblks --;

}

if((size = endpos % SFS\_BLKSIZE) != 0)//计算结束块读取的块长度,如果文件结束位置不对齐

{

if((ret = sfs\_bmap\_load\_nolock(sfs, sin, blkno, &ino)) != 0)//同上

goto out;

if((ret = sfs\_buf\_op(sfs, buf, size, ino, 0)) != 0)

goto out;

alen += size;//同上

}

在进行文件读取时，单次读操作的基本工作单位是块，即单次读操作将加载相应块的所有内容到内存空间。一般情况下，单个文件通常会存储在若干个块上。当文件的起始或结束地址不是块长的倍数，即地址未对齐时，文件首/尾部的部分内容可能只占据相应块的部分存储空间，因此需要进行特殊处理。

接着实现基于文件系统的执行程序机制，具体实现八部分：

1.建立内存管理器

if(current->mm != NULL)//检查进程的内存管理器是否清空

panic("load\_icode: current->mm must be empty.\n");

int ret = -E\_NO\_MEM;

struct mm\_struct \*mm;

if((mm = mm\_create()) == NULL)//建立内存管理器

goto bad\_mm;

2.建立页目录

if(setup\_pgdir(mm) != 0)//设置内存管理器的页目录表

goto bad\_pgdir\_cleanup\_mm;

3.将文件逐段加载到内存中

struct \_\_elfhdr \_\_\_elfhdr\_\_;

struct elfhdr32 \_\_elf, \*elf = &\_\_elf;

if((ret = load\_icode\_read(fd, &\_\_\_elfhdr\_\_, sizeof(struct \_\_elfhdr), 0)) != 0)

goto bad\_elf\_cleanup\_pgdir;

\_load\_elfhdr((unsigned char\*)&\_\_\_elfhdr\_\_, &\_\_elf);

if(elf->e\_magic != ELF\_MAGIC)

{

ret = -E\_INVAL\_ELF;

goto bad\_elf\_cleanup\_pgdir;

}

struct proghdr \_ph, \*ph = &\_ph;

uint32\_t vm\_flags, phnum;

uint32\_t perm = 0;

struct Page \*page;

for (phnum = 0; phnum < elf->e\_phnum; phnum ++)

{

off\_t phoff = elf->e\_phoff + sizeof(struct proghdr) \* phnum;

if((ret = load\_icode\_read(fd, ph, sizeof(struct proghdr), phoff)) != 0)

goto bad\_cleanup\_mmap;

if(ph->p\_type != ELF\_PT\_LOAD)

continue ;

if(ph->p\_filesz > ph->p\_memsz)

{

ret = -E\_INVAL\_ELF;

goto bad\_cleanup\_mmap;

}

vm\_flags = 0;

perm |= PTE\_U;

if(ph->p\_flags & ELF\_PF\_X)

vm\_flags |= VM\_EXEC;

if(ph->p\_flags & ELF\_PF\_W)

vm\_flags |= VM\_WRITE;

if(ph->p\_flags & ELF\_PF\_R)

vm\_flags |= VM\_READ;

if(vm\_flags & VM\_WRITE)

perm |= PTE\_W;

if((ret = mm\_map(mm, ph->p\_va, ph->p\_memsz, vm\_flags, NULL)) != 0)

goto bad\_cleanup\_mmap;

off\_t offset = ph->p\_offset;

size\_t off, size;

uintptr\_t start = ph->p\_va, end, la = ROUNDDOWN\_2N(start, PGSHIFT);

end = ph->p\_va + ph->p\_filesz;

while (start < end)

{

if((page = pgdir\_alloc\_page(mm->pgdir, la, perm)) == NULL)

{

ret = -E\_NO\_MEM;

goto bad\_cleanup\_mmap;

}

off = start - la, size = PGSIZE - off, la += PGSIZE;

if(end < la)

size -= la - end;

if((ret = load\_icode\_read(fd, page2kva(page) + off, size, offset)) != 0)

goto bad\_cleanup\_mmap;

fence\_i(page2kva(page)+off, size);

start += size, offset += size;

}

end = ph->p\_va + ph->p\_memsz;

if(start < la)

{

if(start >= end)

continue ;

off = start + PGSIZE - la, size = PGSIZE - off;

if(end < la)

size -= la - end;

memset(page2kva(page) + off, 0, size);

fence\_i(page2kva(page) + off, size);

start += size;

assert((end < la && start == end) || (end >= la && start == la));

}

while (start < end) {

if((page = pgdir\_alloc\_page(mm->pgdir, la, perm)) == NULL)

{

ret = -E\_NO\_MEM;

goto bad\_cleanup\_mmap;

}

off = start - la, size = PGSIZE - off, la += PGSIZE;

if(end < la)

size -= la - end;

memset(page2kva(page) + off, 0, size);

fence\_i(page2kva(page) + off, size);

start += size;

}

}

sysfile\_close(fd);

4. 建立相应的虚拟内存映射表

vm\_flags = VM\_READ | VM\_WRITE | VM\_STACK;//设置用户栈的权限,将用户栈所在的虚拟内存区域设置为合法

if((ret = mm\_map(mm, USTACKTOP - USTACKSIZE, USTACKSIZE, vm\_flags, NULL)) != 0)

goto bad\_cleanup\_mmap;

5. 建立并初始化用户堆栈

mm\_count\_inc(mm);//切换到用户进程空间

current->mm = mm;

current->cr3 = PADDR(mm->pgdir);

lcr3(PADDR(mm->pgdir));

6. 处理用户栈中传入的参数

uintptr\_t stacktop = USTACKTOP - argc \* PGSIZE;

char \*\*uargv = (char \*\*)(stacktop - argc \* sizeof(char \*));

for (int i = 0; i < argc; i ++)

uargv[i] = strcpy((char \*)(stacktop + i \* PGSIZE), kargv[i]);

7. 设置用户进程的中断帧

struct trapframe \*tf = current->tf;

memset(tf, 0, sizeof(struct trapframe));

tf->tf\_era = elf->e\_entry;

tf->tf\_regs.reg\_r[LOONGARCH\_REG\_SP] = USTACKTOP;

uint32\_t status = 0;

status |= PLV\_USER;

status |= CSR\_CRMD\_IE;

tf->tf\_prmd = status;

tf->tf\_regs.reg\_r[LOONGARCH\_REG\_A0] = argc;

tf->tf\_regs.reg\_r[LOONGARCH\_REG\_A1] = (uint32\_t)uargv;

8. 错误处理

ret = 0;//函数正常退出

out:

return ret;

bad\_cleanup\_mmap:

panic("bad\_cleanup\_mmap");

exit\_mmap(mm);

bad\_elf\_cleanup\_pgdir:

panic("bad\_elf\_cleanup\_pgdir");

put\_pgdir(mm);

bad\_pgdir\_cleanup\_mm:

panic("bad\_pgdir\_cleanup\_mm");

mm\_destroy(mm);

bad\_mm:

panic("bad\_mm");

goto out;

其中需要注意的点在与第三部分中，在物理内存空间分配完成后，读取指定大小的磁盘块，并将读入的数据存储到分配的空间中。当BSS段(存放未初始化或初始化为0的全局变量和静态局部变量)存在，并且先前的TEXT/DATA段(存放代码/初始化不为0的全局变量和静态局部变量)分配的最后一页未被完全占用，则BSS段在进行清零初始化后占用剩余部分，当空间不足时则进一步分配。

**2.6 I/O管理**

数据传输是磁盘I/O操作的核心。读操作将数据从磁盘块设备读取到内存缓冲区，写操作将数据从内存缓冲区写入磁盘块设备。数据传输通常通过DMA（直接内存访问）或PIO（编程I/O）方式进行。DMA方式可以减少CPU的负担，提高传输效率。缓冲区管理是指在内存中分配、使用和释放数据缓冲区的过程。在磁盘I/O操作中，缓冲区管理至关重要，可以提高数据传输的效率和可靠性。操作系统通常会维护一个缓冲区池，用于存储正在传输的数据。ucore利用了iobuf（输入/输出缓冲区）数据结构。iobuf的主要作用是作为内存和设备之间的数据桥梁，管理数据的传输过程。

首先要求实现io缓冲区传输操作

iobuf\_move(struct iobuf \*iob, void \*data, size\_t len, bool m2b, size\_t \*copiedp) {

size\_t alen;

if ((alen = iob->io\_resid) > len) {

alen = len;

}

if (alen > 0) {

void \*src = iob->io\_base, \*dst = data;

if (m2b) {

void \*tmp = src;

src = dst, dst = tmp;

}

memmove(dst, src, alen);

iobuf\_skip(iob, alen), len -= alen;

}

if (copiedp != NULL) {

\*copiedp = alen;

}

return (len == 0) ? 0 : -E\_NO\_MEM;

}

iobuf\_move()函数首先确定实际可以移动的数据长度alen，并根据m2b标志决定数据的移动方向。接着，通过memmove()函数将数据从源缓冲区复制到目标缓冲区，然后调用iobuf\_skip()函数更新iobuf的偏移量和剩余数据量，最后返回实际移动的数据长度。

接着实现块设备的读写操作

disk0\_io(struct device \*dev, struct iobuf \*iob, bool write) {

off\_t offset = iob->io\_offset;

size\_t resid = iob->io\_resid;

uint32\_t blkno = offset / DISK0\_BLKSIZE;

uint32\_t nblks = resid / DISK0\_BLKSIZE;

/\* don't allow I/O that isn't block-aligned \*/

if ((offset % DISK0\_BLKSIZE) != 0 || (resid % DISK0\_BLKSIZE) != 0) {

return -E\_INVAL;

}

/\* don't allow I/O past the end of disk0 \*/

if (blkno + nblks > dev->d\_blocks) {

return -E\_INVAL;

}

/\* read/write nothing ? \*/

if (nblks == 0) {

return 0;

}

lock\_disk0();

while (resid != 0) {

size\_t copied, alen = DISK0\_BUFSIZE;

if (write) {

iobuf\_move(iob, disk0\_buffer, alen, 0, &copied);

assert(copied != 0 && copied <= resid && copied % DISK0\_BLKSIZE == 0);

nblks = copied / DISK0\_BLKSIZE;

disk0\_write\_blks\_nolock(blkno, nblks);

}

else {

if (alen > resid) {

alen = resid;

}

nblks = alen / DISK0\_BLKSIZE;

disk0\_read\_blks\_nolock(blkno, nblks);

iobuf\_move(iob, disk0\_buffer, alen, 1, &copied);

assert(copied == alen && copied % DISK0\_BLKSIZE == 0);

}

resid -= copied, blkno += nblks;

}

unlock\_disk0();

return 0;

}

disk0\_io 函数负责处理块设备的读写请求。实现该函数时，需要遵循以下步骤：

首先，计算起始块号 blkno 和需要传输的块数量 nblks。

然后，进行块对齐和边界检查，确保I/O操作合法。

接下来，使用 iobuf\_move 函数在 I/O 缓冲区和磁盘缓冲区之间移动数据，并调用相应的磁盘读写函数（如 disk0\_write\_blks\_nolock 和 disk0\_read\_blks\_nolock）。

最后，更新 resid 和 blkno，以反映传输的进度，并使用锁机制 lock\_disk0 和 unlock\_disk0 确保在多线程环境下操作的原子性和一致性。

最后一步实现中断的使能与禁用：

void

intr\_enable(void) {

\_\_csrxchg(LISA\_CSR\_CRMD\_IE, LISA\_CSR\_CRMD\_IE, LISA\_CSR\_CRMD);

}

/\* intr\_disable - disable irq interrupt \*/

void

intr\_disable(void) {

\_\_csrxchg(0, LISA\_CSR\_CRMD\_IE, LISA\_CSR\_CRMD);

}

这段代码实现了LoongArch架构下的中断控制基本功能。通过操作处理器的CRMD(Current Mode)控制寄存器中的IE(Interrupt Enable)位来实现中断的开启与关闭。其中intr\_enable()函数通过设置IE位为1来使能中断，而intr\_disable()函数则通过将IE位清零来禁用中断。代码使用\_\_csrxchg指令来保证这些操作是原子的，这对于维护系统的中断状态一致性非常重要。这两个函数构成了操作系统中断处理机制的基础，常用于保护临界区和实现同步机制。

**2.7 磁盘调度**

在本节实验中，我们主要实现了基于LOOK算法的磁盘调度器，它基于磁头当前位置和请求队列中的请求来决定磁头移动的方向。首先，LOOK算法会根据当前磁头的方向，向该方向扫描并处理所有请求，直到扫描到最远的请求。当没有请求可以继续服务时，磁头会反向移动并继续处理剩余的请求。这种算法减少了磁头的往返移动，提高了磁盘的访问效率。与SCAN算法不同，LOOK算法在扫描到最远请求后会立即改变方向，而不必扫描整个磁盘。

为了实现LOOK算法调度器，通过遍历链表返回遍历顺序数组，数组中成员为BIO的逻辑块地址。在插入节点时，我们从当前磁头位置开始遍历链表，找到适当的位置插入新的请求节点。在遍历链表时，我们使用LOOK调度算法按以下步骤进行：

初始化遍历方向：根据direction\_clook决定初始遍历方向（从头到尾或从尾到头）。

遍历链表：按当前方向遍历请求节点，将每个节点的块号记录到traversed\_blocks数组中，并更新当前磁头位置。

切换方向：当到达链表末尾时，切换遍历方向，继续遍历剩余的请求节点。

终止条件：当在两个方向上都没有未处理的请求时，结束遍历。

int\* look\_schedule(int \*begin\_pos, int \*begin\_direction, int \*size) {

struct bio\* current;

int\* traversed\_blocks = (int\*)kmalloc(request\_count \* sizeof(int));

int count = 0;

\*begin\_pos = disk\_head\_look->block\_num;

\*begin\_direction = direction\_look;

kprintf("begin\_pos = %d, begin\_direction = %d\n", \*begin\_pos, \*begin\_direction);

while (1) {

if (direction\_look == 1) {

current = disk\_head\_look->next;

} else {

current = disk\_head\_look->prev;

}

while (current->block\_num != -1) {

traversed\_blocks[count++] = current->block\_num;

kprintf("Servicing block: %d\n", current->block\_num);

disk\_head\_look->block\_num = current->block\_num;

struct bio\* temp = current;

if (direction\_look == 1) {

current = current->next;

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

} else {

current = current->prev;

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

kfree(temp);

request\_count--;

}

direction\_look = -direction\_look;

if (request\_count == 0) {

break;

}

}

request\_count = 0;

\*size = count;

return traversed\_blocks;

}

实现了调度算法后，需要统计调度一组磁盘I/O请求的读写时间。在计算磁盘读写时间时，需要考虑以下几个因素：

跨越磁道时间：每跨越一个磁道需要固定的时间。

跨越扇区时间：每跨越一个扇区需要固定的时间。

数据块传输时间：传输一个数据块所需的时间。

同时逻辑块地址（LBA）是一个线性地址，在时间计算时，需要转换为磁盘的物理地址（即柱面、磁道和扇区）。具体实现代码：

static int calculate\_track\_to\_track\_time(int current\_block, int next\_block) {

// 当前块和目标块的柱面号计算

int current\_cylinder = current\_block / (SPT \* HPC);

int next\_cylinder = next\_block / (SPT \* HPC);

// 计算柱面距离（绝对值）

int track\_distance = abs(current\_cylinder - next\_cylinder);

// 返回总时间：距离乘以单位跨柱面时间

return track\_distance \* TRACK\_SEEK\_TIME;

return 0;

}

// 计算跨扇区的寻道时间

static int calculate\_sector\_to\_sector\_time(int current\_block, int next\_block) {

// 当前块和目标块的扇区号计算

int current\_sector = current\_block % SPT;

int next\_sector = next\_block % SPT;

// 计算扇区距离（绝对值）

int sector\_distance = abs(current\_sector - next\_sector);

// 规范化扇区距离（考虑循环）

int normalized\_distance = min(sector\_distance, SPT - sector\_distance);

// 返回总时间：距离乘以单位跨扇区时间

return normalized\_distance \* SECTOR\_SEEK\_TIME;

return 0;

}

// 计算调度过程中所有请求的总时间

static int measure\_time(int\* traversed\_blocks, int size, int begin\_pos, int direction) {

int total\_time = 0; // 总时间初始化为0

int current\_block = begin\_pos; // 设置初始块号为开始位置

// 遍历所有经过的块

for (int i = 0; i < size; i++) {

int next\_block = traversed\_blocks[i]; // 下一个块号

// 计算跨磁道时间并累加

total\_time += calculate\_track\_to\_track\_time(current\_block, next\_block);

// 计算跨扇区时间并累加

total\_time += calculate\_sector\_to\_sector\_time(current\_block, next\_block);

// 数据块传输时间累加

total\_time += TIME\_DATA\_BLOCK\_TRANSFER;

// 更新当前块号为目标块号

current\_block = next\_block;

}

return total\_time; // 返回总时间

return 0;

}

跨越磁道时间指的是访问不同柱面的块需要跨越磁道的时间。在这里考虑磁盘单向旋转，像 9824 逻辑块到 15522 逻辑块，虽然 9824 在 24 扇区，15522 在 22 扇区，但是由于单向旋转，所以说这种就算的是跨越了（22-24+50=48）扇区。