LA32-uCore操作系统设计

项目成员：

姓名 联系方式（QQ）

屈瑾瑜 971589489

指导老师

陈咸彰、何静媛

参考文献

龙芯架构32位精简版参考手册\_v1.02

一、项目简介

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（龙芯架构32位）指令集设计开发了一套操作系统实验。项目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。

本项目以实现如下功能为目标：

1、裸机程序：异常与中断

2、内存管理

3、进程管理

4、同步与互斥：管道通信

5、文件系统

6、I/O管理：I/O缓存管理

7、磁盘调度：LOOK调度算法实现

二、设计内容介绍  
1、裸机程序：异常与中断

**异常与中断的定义：**

操作系统需要对计算机系统中的各种外设进行管理，这就需要CPU和外设能够相互通信才行。一般外设的速度远慢于CPU的速度。如果让操作系统通过CPU“主动关心”外设的事件，即采用通常的轮询(polling)机制，则太浪费CPU资源了。所以需要操作系统和CPU能够一起提供某种机制，让外设在需要操作系统处理外设相关事件的时候，能够“主动通知”操作系统，即打断操作系统和应用的正常执行，让操作系统完成外设的相关处理，然后再恢复操作系统和应用的正常执行。在操作系统中，这种机制称为中断机制。中断机制给操作系统提供了处理意外情况的能力，同时它也是实现进程/线程抢占式调度的一个重要基石。但中断的引入会导致对操作系统的理解更加困难。

在LoongArch32架构中，中断属于异常(Exception)的一种，uCore内核目前处理的异常包括以下类型：

中断           (EX\_IRQ,CSR.ESTAT.Ecode=0)

Load操作页无效 (EX\_TLBL,CSR.ESTAT.Ecode=1)

Store操作页无效 (EX\_TLBS,CSR.ESTAT.Ecode=2)

TLB 重填       (EX\_TLBR,CSR.ESTAT.Ecode=31)

指令不存在     (EX\_RI,CSR.ESTAT.Ecode=13)

指令特权等级错误 (EX\_IPE,CSR.ESTAT.Ecode=14)

系统调用       (EX\_SYS,CSR.ESTAT.Ecode=11)

地址错误例外     (EX\_ADE,CSR.ESTAT.Ecode=8) 例如地址没有对齐

LoongArch32架构的处理器也提供了两个例外入口。分别是常规例外与TLB例外。由于TLB例外涉及重填页表的工作，因此必须为物理地址。而常规例外入口则可以根据目前处理器的运行状态选择使用虚拟地址或物理地址。这两个例外入口也存储在CSR寄存器中，名称分别为CSR.EBASE与CSR.RFBASE。uCore对于CSR.RFBASE的处理采用了两层跳转的方式，本次实验我们暂时不考虑RFBASE的处理过程。

**时钟中断：**

LoongArch32的CSR提供了一个时钟定时器，操作系统可以通过CSR指令配置该定时器，从而实现硬件产生时钟中断，使得操作系统能够进行分时调度。

这一部分大家请参考龙芯架构32位精简版参考手册\_v1.02.pdf的 7.6定时器相关控制状态寄存器 ，以及 7.4.5例外状态（ESTAT） 的介绍。

**串口中断：**

我们的QEMU以及Chiplab SoC上都有一个NS16550a兼容的串口控制器，该控制器在很多种情况下都会有中断产生（大家可以自行查阅ns16550有关资料），在本实验中我们暂时只需要知道，串口有输入时会产生中断。

在LoongArch32的QEMU以及Chiplab SoC上，串口连接的中断为HWI[0]。

**操作系统异常响应：**

对于uCore，CSR.EBASE在内核初始化阶段（位于kern/init/init.c的setup\_exception\_vector函数）会被初始化为exception\_handler符号对应的指针（位于kern/trap/exceptions.S）。

在kern/trap/exceptions.S中，会完成以下操作：

* 1. 将t0、t1保存到CSR的KS0、KS1中，以便我们可以使用这两个GPR来存储一些数据而不必破坏例外现场。
  2. 判断异常是否发生在用户态，若是，切换栈为内核栈。
  3. 在栈上开辟足够存储trapframe数据结构的空间（位于kern/trap/loongarch\_trapframe.h），将异常发生现场按照trapframe结构保存在栈中。

注：其中的pushreg结构只保存30个GPR寄存器，而不是32个。

这是因为0号寄存器横为0不可写，因此不需要保存，1号寄存器在ABI中意义是RA，已经保 存在了trapframe中的tf\_ra。

需要注意的是，ra寄存器与CSR.ERA意义不同。ra寄存器在ABI上存储的是程序运行中函数return的返回地址，而CSR.ERA存储的是例外产生时还没提交的PC地址。

* 1. 数据保存后，将a0寄存器保存为指向trapframe的指针

根据ABI定义，a0保存的是函数调用的第一个参数，这样C语言编写的loongarch\_trap函数读取该参数即读取a0的值。

2、内存管理

**页空闲块维护：**

irst\_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，双向链表是一个很好的选择。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。具体来说，是定义了一个名为free\_area\_t的数据结构：



其中free\_area\_t包含了一个list\_entry结构的双向链表指针和记录当前空闲页的个数的无符号整型变量nr\_free。其中的链表指针指向了空闲的物理页。

**空闲页链表初始化：**

default\_init\_memmap函数将根据每个物理页帧的情况来建立空闲页链表，也就是初始化了一个空闲块，空闲块应该是根据地址高低形成一个有序链表。地址从高到低的设计是为了在default\_ free\_pages中实现空闲块的合并。初始化过程中default\_init\_memmap(base, n)会在物理内存中从base开始连续申请n个Page，并对这些Page做初始化，比如将ref设置为0（表示这一表项没有被其他虚拟页表引用），将除第一页的其他Page的property, flag设置为0，表示property属性无效并且非预留，将第一页property设置为n，表示这一个空闲块里面有n个Page。

**块分配：**

default\_alloc\_pages函数从空闲页链表中寻找第一个大小足够的空闲块并分配（first fit）。当想要申请n个Pages，先会判断nr\_free是否大于等于n，如果小于则返回NULL。firstfit需要从空闲链表头开始查找最小的地址，通过list\_next找到下一个空闲块元素，通过le2page宏可以由链表元素获得对应的Page指针p。通过p->property可以了解此空闲块的大小。如果>=n，就找到并返回，如果<n，则list\_next，继续查找。直到list\_next== &free\_list，这表示找完了一遍了。找到后，就要从新组织空闲块，然后把找到的page返回。

**块释放：**

default\_free\_pages函数的实现其实是default\_alloc\_pages的逆过程。先遍历这个块的所有页，确认所有页都合法（reserved==0&&property==0），然后将需要释放的空间标记为空之后，找到空闲表中合适的位置。由于空闲表中的记录都是按照物理页地址排序的，所以如果插入位置的前驱或者后继刚好和释放后的空间邻接，那么需要将新的空间与前后邻接的空间合并形成更大的空间。

3、进程管理

**进程创建过程：**

创建一个用户态进程并加载了应用程序后，CPU让这个应用程序最终在用户态执行起来的全过程如下：

① 调用mm\_create函数申请进程内存管理的数据结构mm所需的内存空间，并对mm初始化；

② 调用setup\_pgdir函数，申请一个页目录表所需的一个页大小的内存空间，并把描述ucore内核虚空间映射的内核页表的内容拷贝到此新目录表中，最后mm->pgdir指向此页目录表，也就是进程新的页目录表了，且能够正确映射内核；

③ 根据应用程序执行码的起始位置来解析此ELF格式的执行程序，并调用mm\_map函数根据ELF格式的执行程序说明的各个段（代码段、数据段、BSS段等）的起始位置和大小建立对应的vma结构，并把vma插入到mm结构中，从而表明了用户进程的合法用户态虚拟地址空间；

④ 调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，在页表中建立好物理地址和虚拟地址的映射关系。然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；

⑤ 需要给用户进程设置用户栈，为此调用mm\_mmap函数建立用户栈的vma结构，明确用户栈的位置在用户虚空间的顶端，大小为256个页，即1MB，并分配一定数量的物理内存且建立好栈的虚地址-物理地址映射关系；

⑥ 至此，进程内的内存管理vma和mm数据结构已经建立完成，于是把mm->pgdir赋值到cr3寄存器中，即更新了用户进程的虚拟内存空间，此时的initproc已经被hello的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；

⑦ 先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让CPU转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断。

至此，CPU让这个应用程序最终以用户态执行起来。

**父进程创建子进程：**

父进程调用fork()系统调用，进入正常的中断处理机制，最终调用syscall()函数；在syscall()函数中，根据系统调用名，调用sys\_fork()函数；该函数进一步调用了do\_fork()函数，这个函数的任务就是创建子进程、并且将父进程的内存空间复制给子进程。

在do\_fork()函数中，调用copy\_mm()进行内存空间的复制；在copy\_mm()函数中，又调用了dup\_mmap()函数，该函数是将父进程的虚拟内存空间的内容复制到子进程的内存空间，其又调用了copy\_range()函数来完成父进程内存空间复制到子进程的功能，也就是我们需要补充的部分。

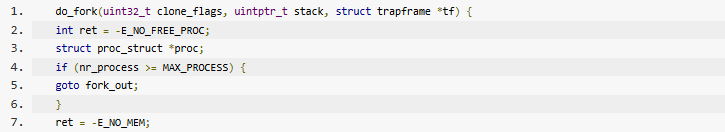
copy\_range()函数遍历父进程指定的某一段内存空间中的每一个虚拟页，如果这个虚拟页是存在的话，为子进程对应的同一个地址也申请分配一个物理页，然后将前者中的所有内容复制到后者中去，然后为子进程的这个物理页和对应的虚拟地址建立映射关系。

**进程相关的函数：**



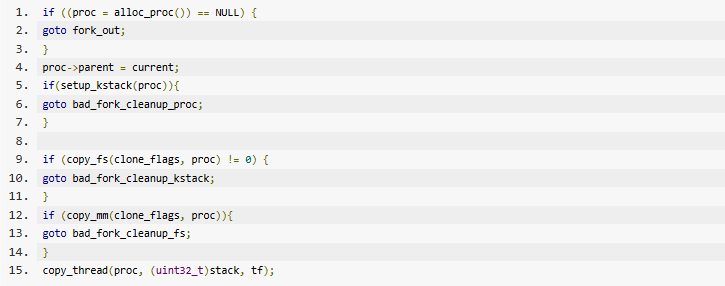
**fork()：**fork的功能是创建一个新进程，具体地说是创建一个新进程所需的控制信息。

该函数的调用过程为：fork->SYS\_fork()->do\_fork()+wakeup\_proc()。首先当程序执行 fork() 时，fork()使用了系统调用SYS\_fork()，而系统调用SYS\_fork()则主要是由do\_fork()和wakeup\_proc()来完成的。



该函数传入的参数有clone\_flags（标志是否进行克隆），stack（记录了分配给该进程/线程的内核栈的位置），tf（中断帧的指针），返回类型为int，值为进程号。

首先给ret赋值-E\_NO\_FREE\_PROC，尝试为进程分配内存，然后使用struct proc\_struct \*proc 定义新进程，再和最大进程数MAX\_PROCESS比较，如果超出则返回到fork\_out代码块



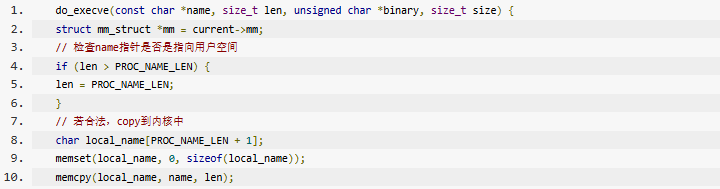
之后调用alloc\_proc() 函数为子进程proc分配内存并初始化进程控制块，将创建出来的子进程proc的parent指针指向当前进程current，调用setup\_stack() 函数进行分配并初始化内核栈，为内核进程建立栈空间。然后调用copy\_fs()函数和copy\_mm()函数根据clone\_flags标志分别复制进程的文件相关信息和进程的内存管理结构。再copy\_thread()函数设置进程在内核(将来也包括用户态)正常运行和调度所需的中断帧（tf）和执行上下文（stack）。



上述操作执行完后，使用get\_pid()为进程分配一个PID，再调用hash\_proc()为进程建立hash映射，set\_links()把设置好的进程控制块放入hash\_list 和 proc\_list 两个全局进程链表中。最后调用wakeup\_proc()函数唤醒该进程并设置其状态为就绪态，并将记录了该进程的进程号proc->pid返回。

**exec()：**exec()的功能是在已经存在的进程的上下文中运行新的可执行文件，替换先前的可执行文件。

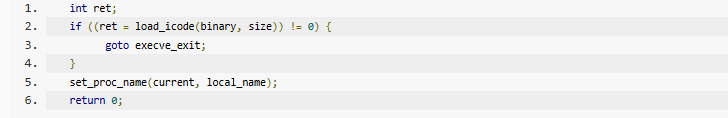
调用过程为： SYS\_exec()->do\_execve()。当应用程序执行的时候，会调用SYS\_exec() 系统调用，而当 ucore 收到此系统调用的时候，则会使用do\_execve()函数来实现。



首先检查用户态虚拟内存空间是否合法（也就是检查name指针是否是指向用户空间），如果合法则执行memset()和memcpy()函数将其复制到内核中。



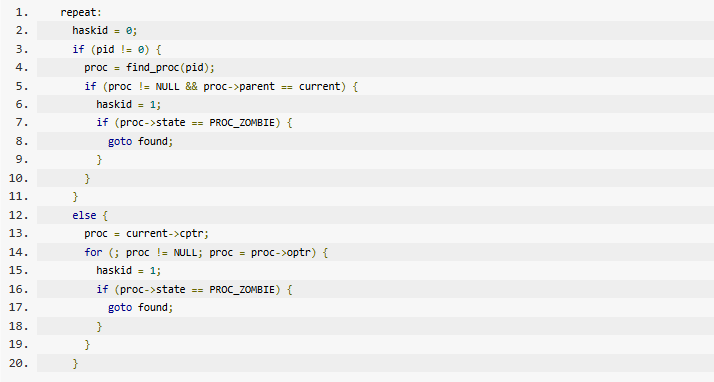
后续判断mm内存管理指针如果不为NULL，则调用lcr3()函数设置页表为内核空间页表，再调用mm\_count\_dec()判断mm的引用计数减1后是否为0，如果为0则表明没有进程再需要此进程所占用的内存空间，则使用exit\_mmap(),mm\_destroy()等函数释放进程所占用户空间内存和进程页表本身所占空间，最后把当前进程的mm内存管理指针置为NULL。



最后调用load\_icode()加载应用程序执行码（binary）到当前进程的新创建的用户态虚拟空间中,调用set\_proc\_name()设置进程的名字。

**wait()：**wait()的功能是等待子进程结束，从而释放子进程占用的资源。

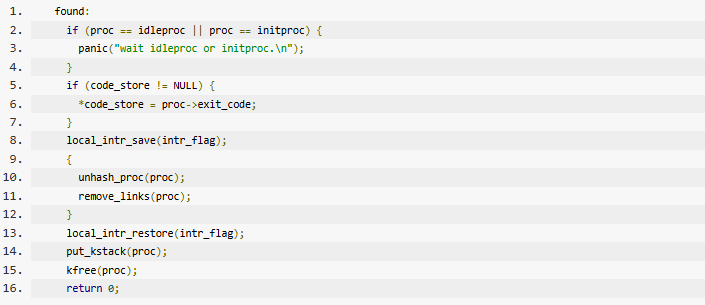
调用过程为： SYS\_wait()->do\_wait()。首先当程序执行 wait()时，wait()使用了系统调用SYS\_wait()，而系统调用SYS\_wait()则主要是由do\_wait()来完成的



在repeat代码块中，第一个if-else判断分支中先判断pid是否为0，如果pid！=0，则使用find\_proc()函数找到进程id为pid的处于退出状态的子进程，如果pid==0，则在for循环中随意找一个处于退出状态的子进程。



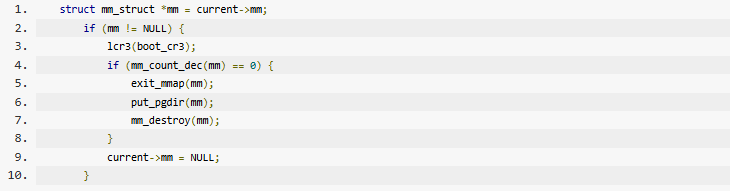
在上面的if-else判断分支中，如果子进程的执行状态为 PROC\_ZOMBIE，都会跳转到found代码块，不会进入到这里的if语句中，因此如果子进程的执行状态不为 PROC\_ZOMBIE，则说明子进程还没有退出，则父进程重新进入睡眠（设置当前状态为PROC\_SLEEPING（睡眠），睡眠原因为WT\_CHILD（等待子进程退出）），调用schedule()函数选择新的进程执行，自己睡眠等待，如果被唤醒则再执行goto repeat进行重复寻找。



如果此子进程的执行状态为 PROC\_ZOMBIE，表明此子进程处于退出状态，此时进入found代码块，使用unhash\_proc()（消除哈希映射）和remove\_links()函数把子进程控制块从两个进程队列proc\_list和hash\_list中删除，并调用put\_kstack()和kfree()释放子进程的内核堆栈和进程控制块。自此，子进程才彻底地结束了它的执行过程，它所占用的所有资源均已释放。

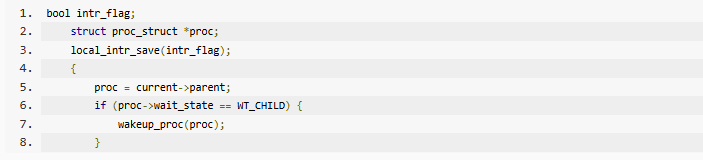
**exit()：**

首先，exit()函数会把一个退出码error\_code传递给ucore，ucore通过执行内核函数do\_exit()来完成对当前进程的退出处理，主要工作就是回收当前进程所占的大部分内存资源，并通知父进程完成最后的回收工作。

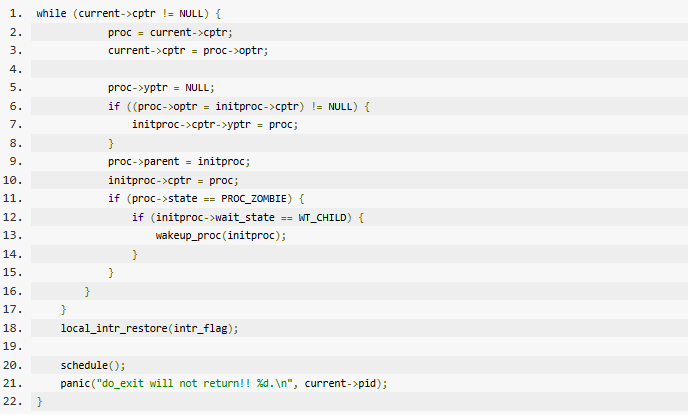


首先判断如果current->mm != NULL，表示是用户进程，则开始回收此用户进程所占用的用户态虚拟内存空间。 a) 首先执行“lcr3(boot\_cr3)”，切换到内核态的页表上，这样当前用户进程目前只能在内核虚拟地址空间执行了，这是为了确保后续释放用户态内存和进程页表的工作能够正常执行。 b) 如果当前进程控制块的成员变量mm的成员变量mm\_count减1后为0（表明这个mm没有再被其他进程共享，可以彻底释放进程所占的用户虚拟空间了。），则开始回收用户进程所占的内存资源。 i. 调用exit\_mmap()函数释放current->mm->vma链表中每个vma描述的进程合法空间中实际分配的内存，然后把对应的页表项内容清空，最后还把页表所占用的空间释放并把对应的页目录表项清空。 ii. 调用put\_pgdir()函数释放当前进程的页目录所占的内存。 iii. 调用mm\_destroy()函数释放mm中的vma所占内存，最后释放mm所占内存。 c) 此时设置current->mm为NULL，表示与当前进程相关的用户虚拟内存空间和对应的内存管理成员变量所占的内核虚拟内存空间已经回收完毕。 1 current->state = PROC\_ZOMBIE; 2 current->exit\_code = error\_code;

这时，设置当前进程的执行状态current->state=PROC\_ZOMBIE，当前进程的退出码current->exit\_code=error\_code。此时当前进程已经不能被调度了，需要此进程的父进程来做最后的回收工作（即回收描述此进程的内核栈和进程控制块）。

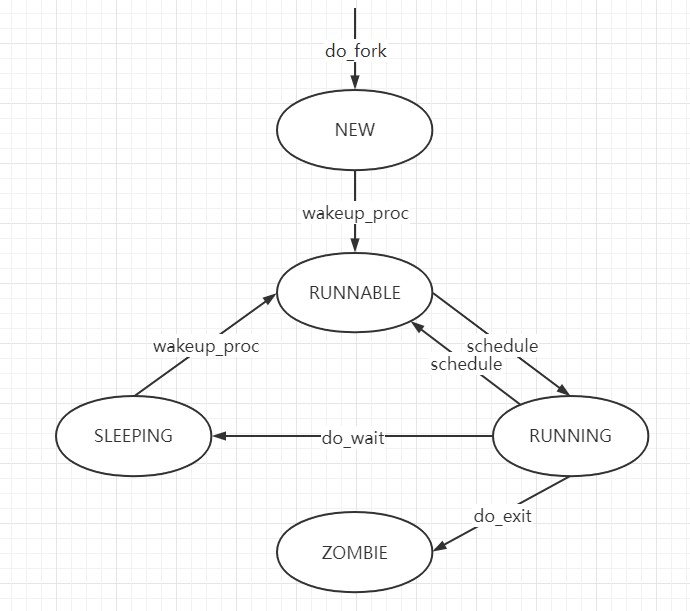


如果当前进程的父进程current->parent处于等待子进程状态：即current->parent->wait\_state==WT\_CHILD，则唤醒父进程（即执行“wakup\_proc(current->parent)”），让父进程帮助自己完成最后的资源回收。



如果当前进程还有子进程，则需要把这些子进程的父进程指针设置为内核线程initproc，且各个子进程指针需要插入到initproc的子进程链表中。如果某个子进程的执行状态是PROC\_ZOMBIE，则需要唤醒initproc来完成对此子进程的最后回收工作。 最后执行schedule()函数，选择新的进程执行。

整体执行关系如下：

  
4、同步与互斥：管道通信

**pipwrite()介绍：**

函数用于将数据写入管道。它的工作流程如下：

1.初始化：w 变量记录已经写入的字节数。

2.循环写入：在循环中，不断将数据从用户缓冲区写入管道，直到所有数据都写入完毕或者遇到阻塞。

3.检查读端是否打开：如果读端关闭（pi->readopen == 0），写操作无效，返回 -1。

4.检查管道是否已满：如果管道已满（pi->nwrite == pi->nread + PIPESIZE），调用 do\_yield() 进行让出处理器时间片，等待空间可用。

5.计算可写入的大小：使用 MIN3 宏计算一次写入的最大字节数，这取决于用户缓冲区剩余大小、管道剩余空间大小以及环形缓冲区的线性剩余容量。

6.写入数据：将数据从用户缓冲区复制到管道缓冲区，并更新写指针 pi->nwrite 和已写入字节数 w。

**pipread()介绍：**

1.初始化：r 变量记录已经读取的字节数。

检查管道是否为空：如果管道为空，且写端打开，调用 do\_yield() 等待数据；如果写端关闭，返回 -1。

2.计算可读取的大小：使用 MIN3 宏计算一次读取的最大字节数，这取决于用户缓冲区剩余大小、管道中可读数据大小以及环形缓冲区的线性剩余容量。

3.读取数据：将数据从管道缓冲区复制到用户缓冲区，并更新读指针 pi->nread 和已读取字节数 r。

**环形缓冲区实现：**

环形缓冲区是一种固定大小的缓冲区，它的读写指针可以在缓冲区的两端循环使用。在上述实现中，pi->nwrite 和 pi->nread 分别是写指针和读指针。通过对这些指针取模操作（% PIPESIZE），实现了环形效果。  
5、文件系统

**load\_icode()函数：**

建立内存管理器

建立页目录

将文件逐段加载到内存中(注意设置虚拟地址与物理地址之间的映射)

建立相应的虚拟内存映射表

建立并初始化用户堆栈

处理用户栈中传入的参数

设置用户进程的中断帧

错误处理  
6、I/O管理：I/O缓存管理

**数据传输与缓冲区管理：**

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。

**Iobuf机制：**

在操作系统中，I/O（输入/输出）操作是核心功能之一，涉及数据在内存和外部设备（如硬盘、SSD等）之间的传输。为了高效地管理这些数据传输，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。它包含以下几个关键字段：

io\_base，指向当前数据缓冲区的起始地址；

io\_offset，指示当前操作的偏移量；

io\_resid，表示剩余的数据量。 iobuf结构共同工作，确保数据在内存和设备之间有效地移动，同时保持操作的正确性和数据的完整性。iobuf\_move函数是一个核心函数，用于在iobuf结构和用户空间或其他缓冲区之间移动数据。这个函数的设计直接影响系统在进行文件读写操作时的性能和效率。在实现iobuf\_move时，需要考虑数据方向、数据完整性和错误处理等因素。

**块设备与字符设备：**

在磁盘I/O操作中，我们主要涉及两种设备：块设备和字符设备。块设备以固定大小的块（通常是512字节或更大）为单位进行数据传输，支持随机访问，即可以直接访问任意一个块的数据。典型的块设备包括硬盘、SSD等。字符设备则以字符为单位进行数据传输，通常不支持随机访问，典型的字符设备包括键盘、串口等。在本实验中，我们主要关注块设备的操作。

**块对齐与边界检查：**

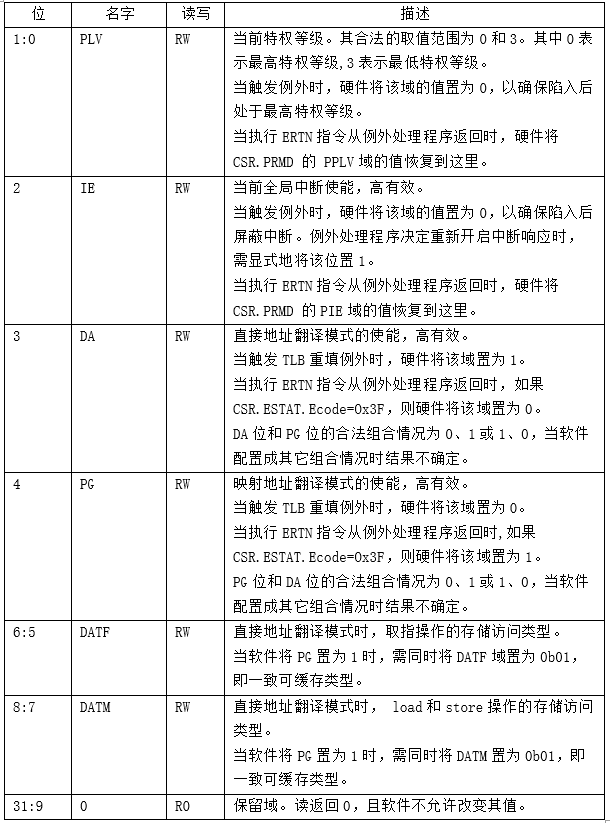
块对齐和边界检查是磁盘I/O操作中的两个重要概念。在进行磁盘I/O操作时，数据的起始地址和长度通常需要是块大小的整数倍，这样可以确保数据的完整性和传输效率。块对齐检查可以防止部分块读写，提高I/O操作的效率。边界检查则确保I/O操作不会超出磁盘的边界，防止数据错误和越界访问。这些检查可以通过简单的取模运算实现，在disk0\_io()函数中通过if ((offset % DISK0\_BLKSIZE) != 0 || (resid % DISK0\_BLKSIZE) != 0) 和 if (blkno + nblks > dev->d\_blocks)语句来判断块对齐和边界对齐。

**中断处理与禁用：**

中断是处理I/O操作中不可或缺的一部分，它允许CPU在处理其他任务时能够响应来自外部设备的请求，进行快速处理。在I/O操作过程中，设备完成数据传输后会产生中断信号，通知CPU进行相应的处理。这种机制可以显著提高系统的效率，因为CPU不需要持续轮询设备状态，从而节省了大量的处理时间和资源。中断处理的优点在于可以快速响应I/O设备的状态变化，提高数据传输的实时性和系统的整体性能。为了深入理解中断在I/O操作中的作用，本实验将引导学生实现中断使能和禁用函数，并通过实际操作验证其功能。通过这一实验，学生将掌握中断的工作原理和实现方法，增强对操作系统中断机制的理解和应用能力。

中断使能函数在intr.c中，函数主要通过\_csrxchg函数来对控制状态器寄存器中的CRMD来进行配置

当前模式信息（CRMD）寄存器定义：



详情请参阅龙芯架构32位精简版参考手册\_v1.02.pdf。由手册可知，CRMD的第2位，控制中断的使能，所以我们可以通过csrxchg函数来对其进行修改，csrxchg是特权指令，它根据通用寄存器rj中存放的写掩码信息，将通用寄存器rd中的旧值写入到指定 CSR中对应写掩码为1的那些比特，该CSR中的其余比特保持不变，同时将该CSR的旧值更新到通用寄存器 rd 中。详情可以查看参考手册。  
7、磁盘调度：LOOK调度算法实现

**链表的基本操作：**

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。具体步骤如下：

初始化指针：从当前磁头位置开始遍历链表。

寻找插入位置：遍历链表，直到找到一个节点，其块号大于或等于要插入的请求块号。

更新指针：将新的请求节点插入到找到的位置，更新相邻节点的指针。

这种方式确保链表中的请求按块号有序排列，有利于之后的LOOK调度算法的遍历。

**LOOK调度算法**

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。

这种方式确保所有请求按照LOOK调度算法的顺序得到处理，并记录遍历顺序以供后续使用。

**磁盘LBA到扇区、磁道、柱面的换算：**

逻辑块地址（LBA）是一个线性地址，需要转换为磁盘的物理地址（即柱面、磁道和扇区）。以下是换算公式：

扇区号 = LBA % SPT（每个柱面扇区数）

磁道号 = (LBA / SPT) % HPC（每个柱面柱面数）

柱面号 = LBA / (SPT \* HPC)

SPT = 每个柱面扇区数

HPC = 每个柱面柱面数

**磁盘读写时间的计算：**

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。

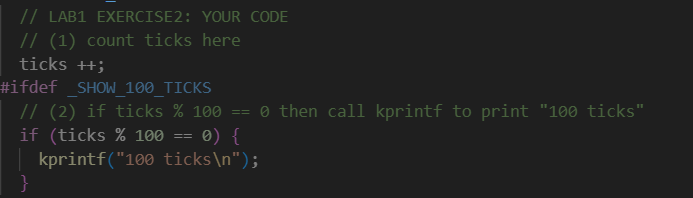
设定时间参数为：

TRACK\_SEEK\_TIME = 跨越磁道时间（单位：毫秒）

SECTOR\_SEEK\_TIME = 跨越扇区时间（单位：毫秒）

TIME\_DATA\_BLOCK\_TRANSFER = 传输一个块数据时间（单位：毫秒）

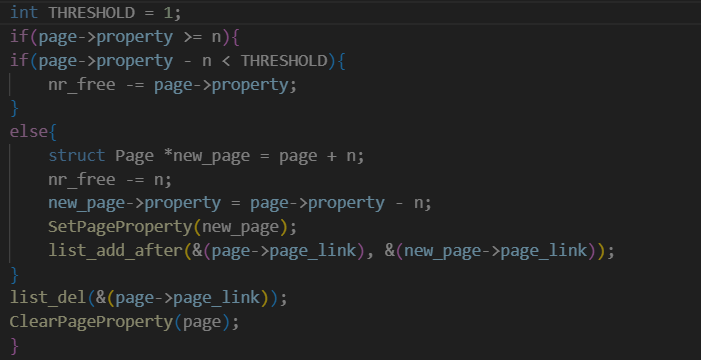
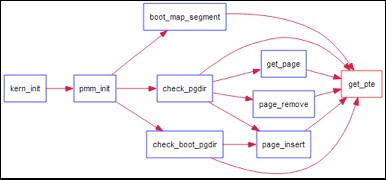
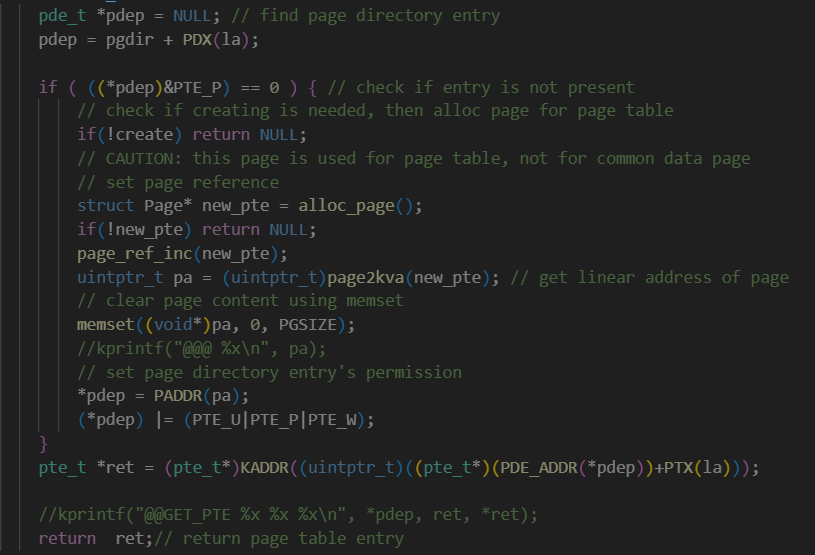
1. 设计内容实现

1、异常与中断  
1.1 实现时钟中断处理函数clock\_int\_handler  
 完善kern/driver/clock.c中的时钟中断处理函数clock\_int\_handler，在对时钟中断进行处理的部分填写trap函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用kprintf，向屏幕上打印一行文字”100 ticks”。  


1.2 实现串口中断处理函数serial\_int\_handler

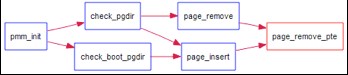
请编程完善kern/driver/console.c中的串口中断处理函数serial\_int\_handler，在接收到一个字符后读取该字符，并调用kprintf输出该字符。  

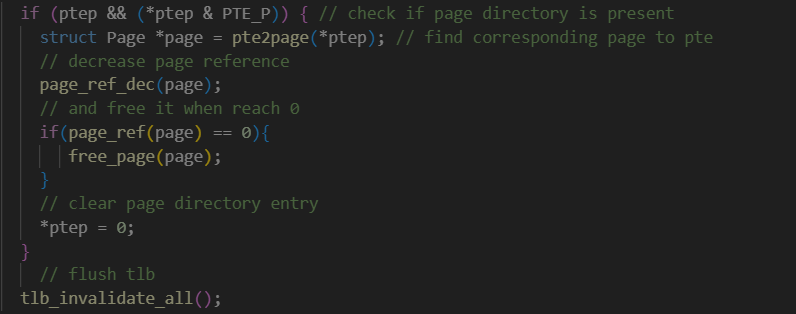

1. 内存管理  
   2.1 实现 first-fit 连续物理内存分配算法

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。  
  
2.2 实现寻找虚拟地址对应的页表项  
 通过设置页表和对应的页表项，可建立虚拟内存地址和物理内存地址的对应关系。其中的get\_pte函数是设置页表项环节中的一个重要步骤。此函数找到一个虚地址对应的二级页表项的内核虚地址，如果此二级页表项不存在，则分配一个包含此项的二级页表。请在仔细阅读和理解get\_pte函数中的注释，编程补全get\_pte函数(kern/mm/pmm.c中)代码，实现其功能。get\_pte函数的调用关系图如下所示：  
  


2.3 释放某虚地址所在的页并取消对应二级页表项的映射

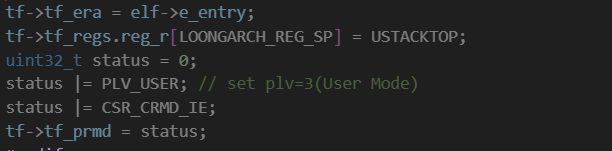
当释放一个包含某虚地址的物理内存页时，需要让对应此物理内存页的管理数据结构Page做相关的清除处理，使得此物理内存页成为空闲；另外还需把表示虚地址与物理地址对应关系的二级页表项清除。请仔细查看和理解page\_remove\_pte函数中的注释，并编程补全在kern/mm/pmm.c中的page\_remove\_pte函数代码。page\_remove\_pte函数的调用关系图如下所示：





3、进程管理  
3.1 第一个用户进程的创建

当一个用户进程创建时，do\_execve()函数调用load\_icode()函数（kern/process/proc.c中）来加载并解析一个处于内存中的ELF执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好proc\_struct结构中的成员变量trapframe中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。



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）。先解释一下代码中变量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。  
3.2 父进程创建子进程  
 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。具体是通过copy\_range函数（位于kern/mm/pmm.c中）实现的，请补充copy\_range()的实现，确保能够正确执行。



编程需要完成的内容就是：

找到父进程需要复制的物理页所在内核虚拟地址；

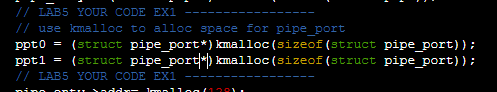
找到子进程需要被填填写的物理页的内核虚拟地址；

将父进程的物理页的内容复制到子进程中去；

建立子进程的物理页与虚拟页的映射。

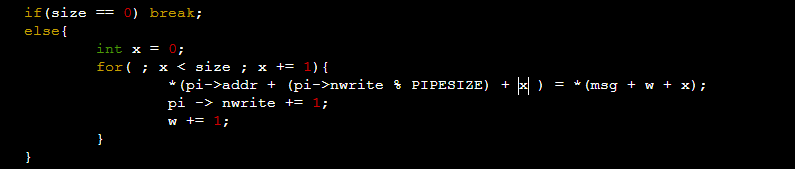
4、同步与互斥  
4.1 完善PIPE相关数据结构和函数

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，在上一个代码块末尾已经声明。



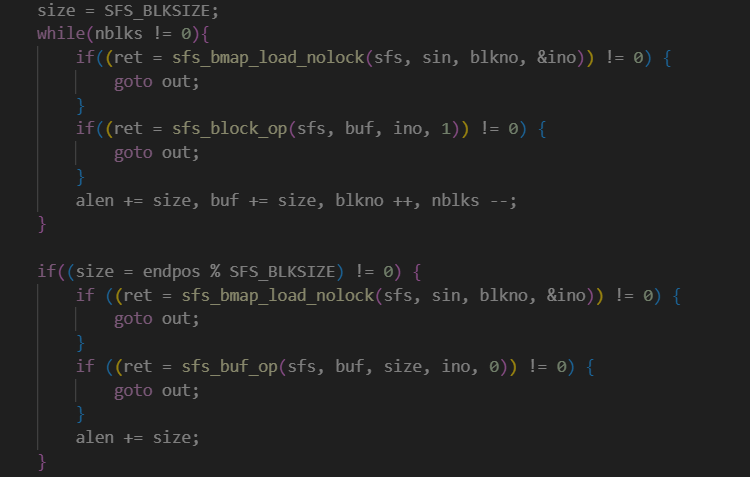
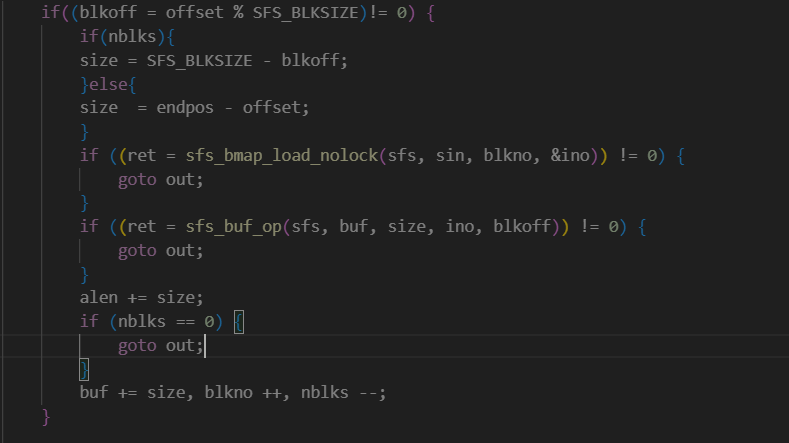
4.2 实现管道通信并进行编译、测试  
 参考piperead函数的实现方式，用for循环对大小为size的空间进行写入，写入地址为\*(pi->addr+w+x)（w是已经写入的空间，x是当前for循环遍历到的位置），写入值为\*(msg+x+w)。

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;



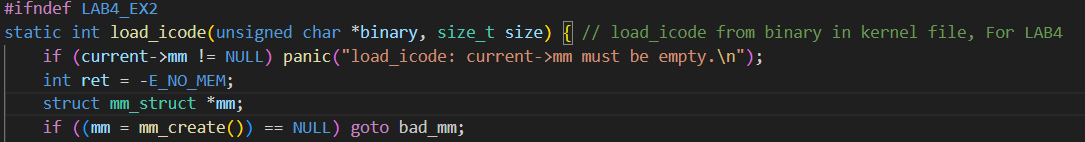
5、文件系统  
5.1 读文件操作的实现

在进行文件读取时，单次读操作的基本工作单位是(盘)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。

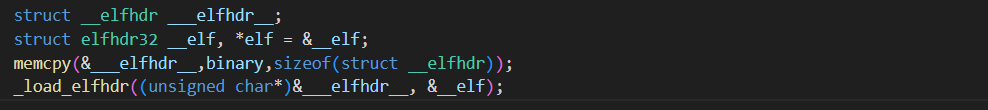


5.2 基于文件系统的执行程序机制的实现

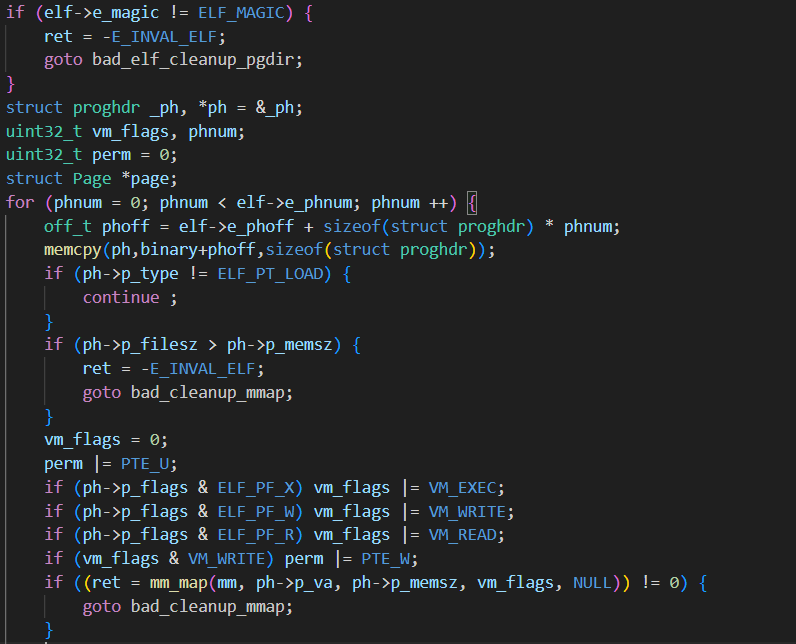
实现load\_icode()函数

1. 建立内存管理器  
   
2. 建立页目录  
   
3. 将文件逐段加载到内存中

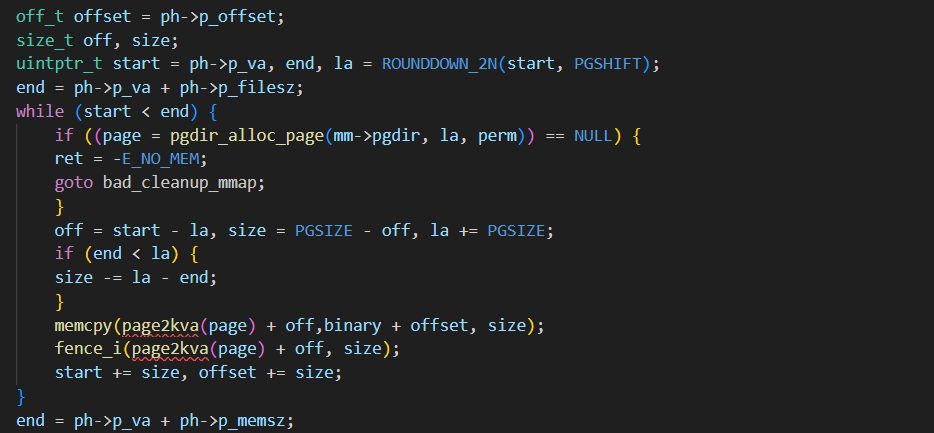
该部分首先将程序从文件加载到内存，从磁盘上读取ELF文件的首部(Elf-header)中的信息，据此循环加载各程序段的程序头信息(Program header)。

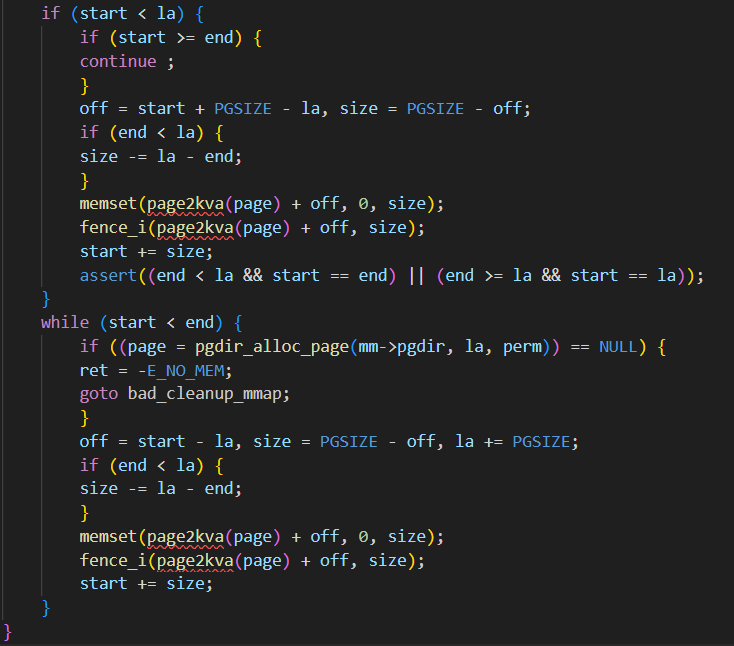


随后，在进行一系列的特判后，建立虚拟地址与物理地址之间的映射，并根据ELF文件中的信息，设置各段的权限，并将对应虚拟内存地址设置为合法。



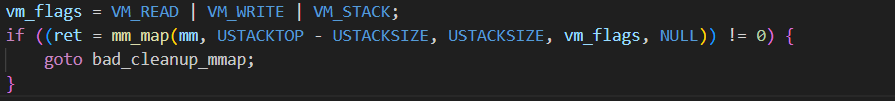
然后复制数据段和代码段。在物理内存空间分配完成后，读取指定大小的磁盘块，并将读入的数据存储到分配的空间中。当BSS段(存放未初始化或初始化为0的全局变量和静态局部变量)存在，并且先前的TEXT/DATA段(存放代码/初始化不为0的全局变量和静态局部变量)分配的最后一页未被完全占用，则BSS段在进行清零初始化后占用剩余部分，当空间不足时则进一步分配。

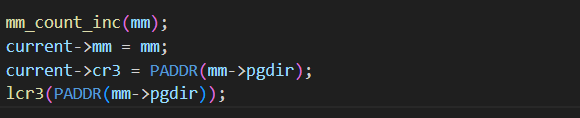


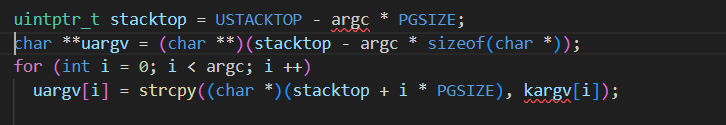


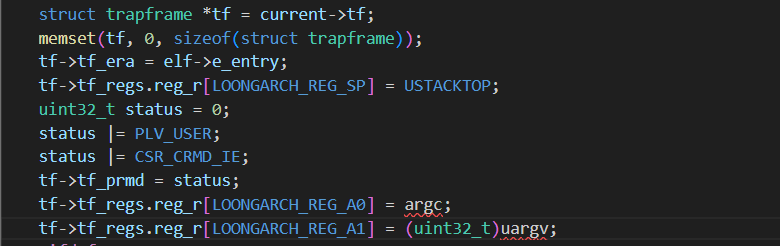
操作完成后，关闭该文件。

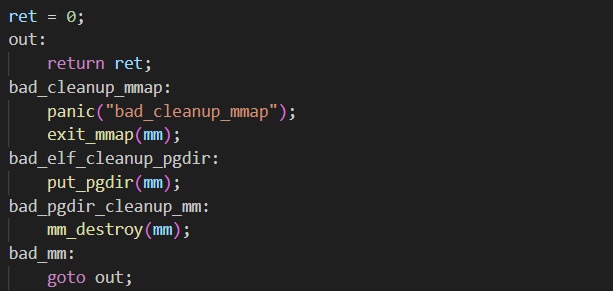
1. 建立相应的虚拟内存映射表



1. 建立并初始化用户堆栈  
   
2. 处理用户栈中传入的参数

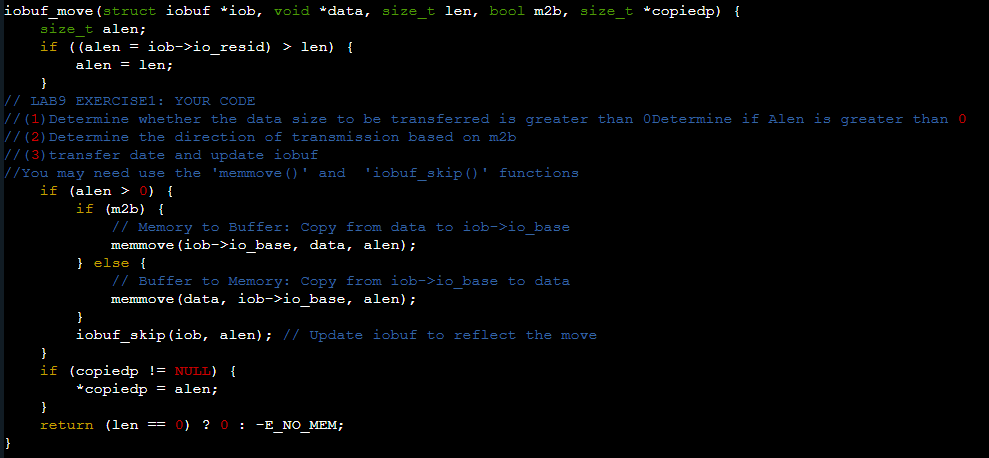


1. 设置用户进程的中断帧  
   
2. 错误处理



1. I/O管理  
   6.1 io缓冲区传输操作的实现

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。



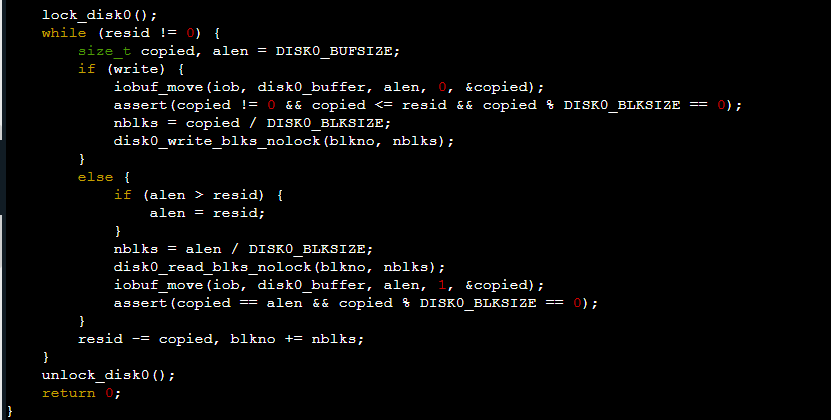
6.2 块设备读写操作的实现

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：

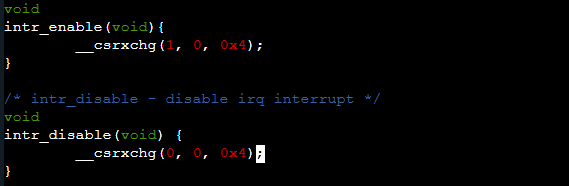
首先，计算起始块号 blkno 和需要传输的块数量 nblks。

然后，进行块对齐和边界检查，确保I/O操作合法。

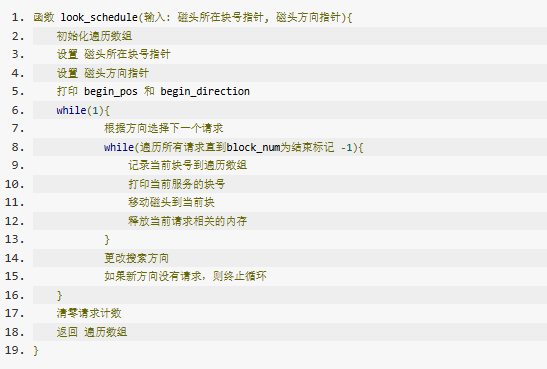
接下来，使用 iobuf\_move 函数在 I/O 缓冲区和磁盘缓冲区之间移动数据，并调用相应的磁盘读写函数（如 disk0\_write\_blks\_nolock 和 disk0\_read\_blks\_nolock）。

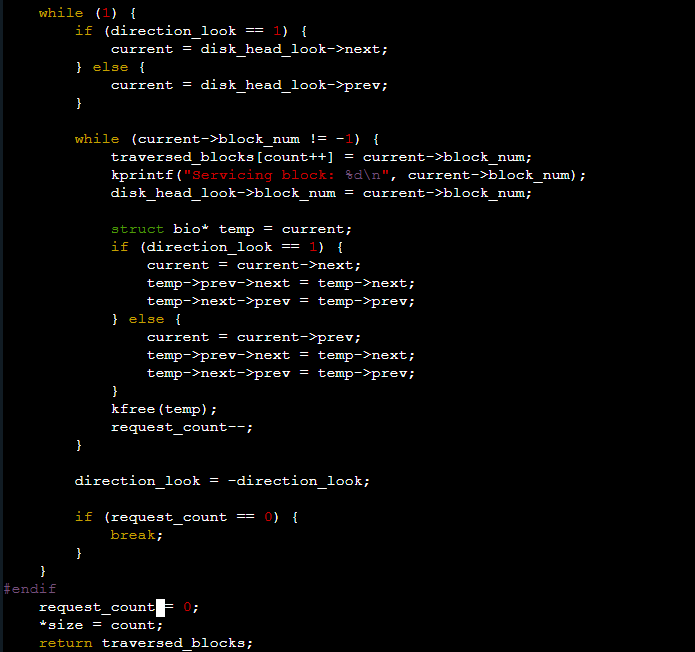
最后，更新 resid 和 blkno，以反映传输的进度，并使用锁机制 lock\_disk0 和 unlock\_disk0 确保在多线程环境下操作的原子性和一致性。  


6.3 中断的使能和禁用

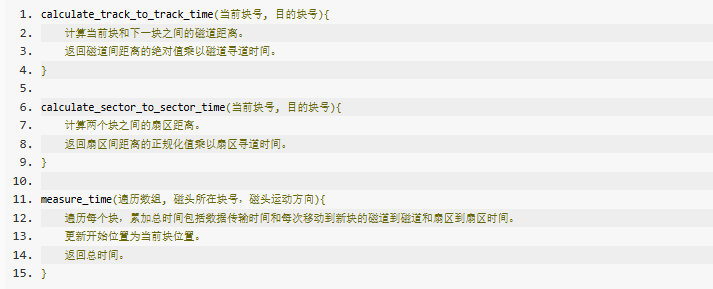


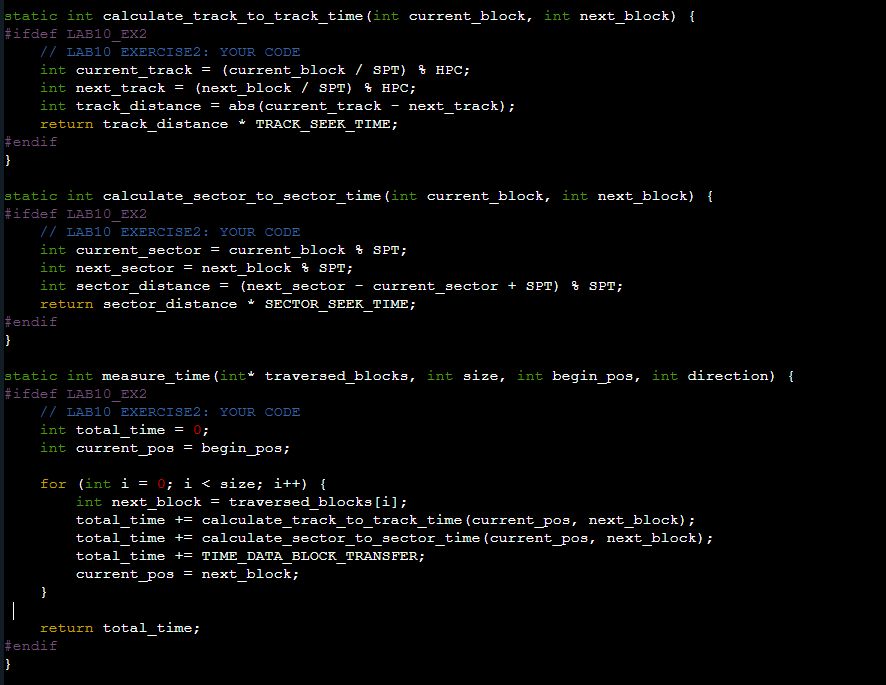
1. 磁盘调度

7.1 实现LOOK算法调度器  
 请补全look\_schedule()代码，实现依据LOOK算法遍历链表，并按照遍历顺序返回数组，数组成员为BIO的逻辑块地址。  




7.2 统计调度一组磁盘I/O请求的读写时间





1. 设计结果

结果：完成了所有设计部分。













