操作系统内核设计赛设计文档（决赛）

成员组成

- 学校：中国科学院大学
- 队名：C-core
- 队员：
  - 王函
  - 裴晓坤
  - 黄天羿

概述

在决赛第二阶段中，我们需要支持**busybox**、**lmbench**和**lua**三种程序的测试用例相关功能，并添加自己选择的应用程序用于展示运行。与决赛第一阶段的**libc-test**相同，这三种程序都基于**musl**库，所以只需要在决赛第一阶段代码的基础上进行系统调用的补充与功能完善、对测试性能的**lmbench**程序进行性能上的优化、以及寻找合适的应用程序进行编译适配即可。

设计实现

系统调用与功能完善

本阶段所需的大部分系统调用在决赛第一阶段中均已被实现，而busybox本身是一个较为复杂的程序，不能够直接通过第一阶段中的方法直接获取使用的系统调用，较大程度上需要通过逐步尝试获取系统调用号。本阶段中，我们需要额外实现的系统调用如下所示：

#define SYS_faccessat 	(48)

ACCESS - Linux手册页-之路教程 (onitroad.com)

#define SYS_sendfile (71)

SENDFILE - Linux手册页-之路教程 (onitroad.com)

#define SYS_pselect6 (72)

SELECT - Linux手册页-之路教程 (onitroad.com)

#define SYS_readlinkat (78)

READLINK - Linux手册页-之路教程 (onitroad.com)

#define SYS_fsync (82)

FSYNC - Linux手册页-之路教程 (onitroad.com)

#define SYS_setitimer (103)

GETITIMER - Linux手册页-之路教程 (onitroad.com)

#define SYS_syslog (116)

SYSLOG - Linux手册页-之路教程 (onitroad.com)

#define SYS_tgkill (131)

TKILL - Linux手册页-之路教程 (onitroad.com)

#define SYS_getrusage (165)

GETRUSAGE - Linux手册页-之路教程 (onitroad.com)

#define SYS_vm86 (166)

VM86 - Linux手册页-之路教程 (onitroad.com)

#define SYS_getuid (174)

GETUID - Linux手册页-之路教程 (onitroad.com)

#define SYS_getgid (176)

GETGID - Linux手册页-之路教程 (onitroad.com)

#define SYS_msync (227)

MSYNC - Linux手册页-之路教程 (onitroad.com)

#define SYS_renameat2 (276)

RENAME - Linux手册页-之路教程 (onitroad.com)

这些系统调用需要实现的功能大多较为直观，可以复用之前的系统调用，而有部分系统调用对本阶段并无太大影响，于是实现较为简单。具体实现可以直接参见实现代码，在此不多加介绍。

除了实现上述系统调用之外，为了实现功能与优化性能，我们还进行了一系列原有功能的调整与完善：

1.将bcache的写回机制调整为write back，同时调整了页面分配机制。

2.删除了share_fd相关机制；将sigaction重新实现为类似fd_table的sig_table进行管理。

3.将大部分原有的静态数组分配内容调整为指针与动态分配的形式，大幅缩小内核大小。

4.完善了lazy mmap机制，并大幅修改了内存回收机制，增加了swap机制与pcb/fd扩充机制。（见后）

5.其他为了实现busybox/lmbench/lua而进行的数不胜数的小调试与改动。（参考提交次数）

本次报告中，我们着重介绍更新的内存回收机制与swap机制。

信号处理机制改进

在第一阶段的比赛当中，musl库中的信号安装函数会设置上restore位。（即下面结构体当中的**restore**字段）

```
typedef struct sigaction{
    void (*handler)(int);
    unsigned long flags;
    void (*restorer)(void);
    sigset_t   sa_mask;
} sigaction_t;
```

在进入用户态的信号处理函数时，我们将restore函数的地址放置到ra寄存器当中，这样信号处理函数执行完毕后，将会自动调用**sigreturn**系统调用恢复进入信号处理函数之前的执行流。但在现有的机制当中，restore字段已经被弃用，因此我们需要采取新的策略，使得信号处理程序执行完毕后将会自动调用信号返回函数。

为了解决这个问题，我们在内核自行编写了一段调用系统调用函数，并将其完整地复制到用户栈的栈顶，并在初始化信号处理时将其地址放置到ra寄存器当中，这样在执行完毕相应的信号处理函数之后，便可以直接跳转到栈顶执行信号返回函数。

Swap换页机制实现

在第二阶段的测试当中，受到k210开发板本身的内存限制，某些程序无法正常执行，为此我们专门设计了完善的swap换页机制。对于换页机制而言，如果我们用数据结构记录每一个进程所使用的页面的信息，将会造成很大的麻烦，因此采用换页机制势在必行。下面对其做简要介绍。

重要数据结构及其字段说明

对于K210上有限的8MB内存而言，使用堆区动态分配代价过于沉重。在以往比赛阶段的报告中，已经详细地介绍了使用结构体，以页面为单位管理所有内存的方法，并将其进行链表化以加快页面分配速度。数据结构如下：

```
typedef struct page_node{
    list_node_t list;
    volatile PTE * page_pte;
    uint64_t kva_begin; /* the va of the page */
    uint64_t sd_sec;    /* last time in SD place */
    uint8_t share_num;  /* share page number */
    uint8_t is_share;   /* maybe for the shm_pg_get */
}page_node_t;
```

为了支持换页机制，我们向数据结构中添加了两个成员变量：

- **page_pte**：其为当前占有该页面的进程的页表项，对应其三级页表中指向该页面的最后一级页表项。通过对这个字段的维护，我们可以知道该页面的如下标志位：

```
#define _PAGE_PRESENT (1 << 0)
#define _PAGE_READ (1 << 1)     /* Readable */
#define _PAGE_WRITE (1 << 2)    /* Writable */
#define _PAGE_EXEC (1 << 3)     /* Executable */
#define _PAGE_USER (1 << 4)     /* User */
#define _PAGE_GLOBAL (1 << 5)   /* Global */
#define _PAGE_ACCESSED (1 << 6) /* Set by hardware on any access \
                                   */
#define _PAGE_DIRTY (1 << 7)    /* Set by hardware on any write */
#define _PAGE_SOFT (1 << 8)     /* Reserved for software */
#define _PAGE_SD   (1 << 9)     /* Clock SD flag */
```

这样我们便可以得知该页面是否为脏页，以及最近是否访问过等信息。

- **sd_sec**：该页面所在的扇区，如果该进程刚从SD卡中换出，则其为刚换出的扇区号，如果为0，则表示其没有被换入磁盘。为了实现的方便，我们没有在SD卡当中建立虚拟文件作为交换区，而是直接指定换页机制的起始扇区，进行向上增长。

这样，我们将需要换页的页面加入到一个双向链表队列当中，便可进行下一步的操作。

改进的clock换页算法

我们注意到，RISCV的页表项中有 _PAGE_ACCESSED 和 _PAGE_DIRTY位， 表示该页面是否被访问以及是否被写入。我们需要充分利用这些标志位（以上提及到的page_pte中实时保存了这些信息）实现改进的clock算法。该算法主要进行四个步骤：

1. **寻找最近没有访问且没有写入的页面**

即寻找 _PAGE_ACCESSED 与 _PAGE_DIRTY 都没有置位的页面，代码如下：

```
// we will find the _PAGE_ACCESSED == 0 and _PAGE_DIRTY == 0 page
page_node_t * find_swap_AD()
{
    list_node_t* entry = clock_pointer;
    list_node_t* q = clock_pointer;
    page_node_t *page_entry;
    page_node_t *swapPage = NULL;
    do
    {
        page_entry = list_entry(q, page_node_t, list);
        // find _PAGE_ACCESSED == 0 and _PAGE_DIRTY == 0
        if (page_entry->page_pte == NULL) goto goon;
        if (!check_attribute(page_entry->page_pte, _PAGE_ACCESSED | _PAGE_DIRTY)
&& \
             check_attribute(page_entry->page_pte, _PAGE_WRITE))
        {
            swapPage = page_entry;
            return swapPage;
        }
goon:   ;
        // we can't stand in the list_head swapPageList
        if (page_entry->list.next == &swapPageList)
            q = page_entry->list.next->next;
        else
            q = page_entry->list.next;
    } while (q != entry);
```

```
    return NULL;
}
```

如果找到了则可将找到的页面作为换出的页面，否则需要进行下一步。

2. **寻找最近没有访问的页面**

即寻找 _PAGE_ACCESSED 位为0的页面，代码如下：

```
// we will find the _PAGE_ACCESSED == 0
page_node_t * find_swap_A()
{
    list_node_t* entry = clock_pointer;
    list_node_t* q = clock_pointer;
    page_node_t *page_entry;
    page_node_t *swapPage = NULL;
    do
    {
        page_entry = list_entry(q, page_node_t, list);
        if (page_entry->page_pte == NULL) goto goon;
        // find _PAGE_ACCESSED == 0
        if (!check_attribute(page_entry->page_pte, _PAGE_ACCESSED) && \
             check_attribute(page_entry->page_pte, _PAGE_WRITE))
        {
            swapPage = page_entry;
            return swapPage;
        } else
            clear_attribute(page_entry->page_pte, _PAGE_ACCESSED);
goon:
        // we can't stand in the list_head swapPageList
        if (page_entry->list.next == &swapPageList)
            q = page_entry->list.next->next;
        else
            q = page_entry->list.next;
    } while (q != page_entry);

    return NULL;
}
```

注意此处需要将查询过的页面的_PAGE_ACCESSED位清空。如果找到则返回，否则继续下一步操作。

3. **再次寻找最近没有访问且没有写入的页面**

进行与第一步一样的操作，如果找到则返回，否则继续进行下一步操作

4. **再次寻找最近没有访问的页面**

进行与第二步一样的操作，最后一定能找到一个替换的页面，因为如果第二步没有找到，则代表着所有的页面的_PAGE_ACCESSED位都为0。

被写回SD卡的页面信息记录

判断页面被写回SD卡

如何判断一个进程的页面是否已被写回SD卡呢？首先我们必须要得知该页面的PTE表项，该信息由数据结构的page_pte进行保存，这里看一下riscv的页表项置位信息：

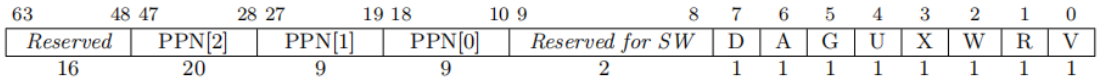

Figure 4.16: Sv39 page table entry.

我们注意到，[9:8]为页面保留位，因此我们直接利用第8位作为SD卡标志位，当页面被换出之后，将V为置为0，并置位SD卡标志位。

记录被写回的页面在SD卡的位置

前面已经提及到，如果用数据结构保存进程被替换的页面的信息，将会极其占用内存，因此这里我们采用一个精巧的设计，将页面保存的扇区号保存到[63:9]当中（完全足够使用）。

将页面换出到SD卡的流程

1. 将PTE的SD卡标志位置为1，V位置为0。
2. 判断相应的结构体当中的sd_sec信息，如果其不为0，则代表该页面曾经被存储到SD卡。这里进行一个判断：
    - 如果该页面 _PAGE_DIRTY 位为0，说明其在取出SD卡之后没有被写入过，这样我们便不需要再将其写回SD卡。（经过我们的测试，这样可以提高几倍的速度！）
    - 如果该页面 _PAGE_DIRTY 位为1，则代表需要将其写回。为了便于管理，将其写入到SD卡上的sd_sec号扇区。
3. 如果sd_sec为0，则需要为其分配扇区号存储。
4. 将sd_sec信息写入到页表项当中。

将页面从SD卡换出的流程

1. 因为其V位为0，当访问这块页面时，必定触发缺页中断，随后根据SD卡位判断其在SD卡当中。
2. 从页表项中取出该页存放在SD卡上的对应位置。
3. 在内存中准备一块页面用于接受该页。
4. 置位V位，清空SD位，并在对应页面的数据结构中记录其刚刚被换出的扇区号（sd_sec）。
5. 如果是Load缺页中断，则将 _PAGE_ACCESSED 位置1，否则将 _PAGE_DIRTY 位置1。

以上便是换页机制的简单介绍。

编译适配应用程序

本次比赛要求我们在完成三个主办方给出的测试程序之外挑选一些合适的应用程序进行交叉编译，并在qemu上基于我们的操作系统运行。本次我们选择了三个应用程序：

1.常用的压缩应用程序**p7zip**。

2.编译链中的常用工具**binutils**。（以readelf为例）

3.我们自己编写的启发式算法五子棋小程序**wzq**。

应用程序的源码放置在根目录的app_sourcecode文件夹下。考虑到p7zip所需的内存占用空间稍大，将qemu的内存大小调整为16MB。经过一些针对性的微调之后，程序能够成功运行，效果如下：

```
> info: the beginning of the OScomp!
1: exec busybox touch test.txt
2: exec busybox echo "hello world" > test.txt
3: exec busybox cat test.txt
hello world
4: exec 7za a -mmt=1 test.7z test.txt
```

```
7-Zip (a) 21.07 (LE) : Copyright (c) 1999-2021 Igor Pavlov : 2021-12-26
 64-bit locale=C UTF8=- Threads:2

Scanning the drive:
1 file, 12 bytes (1 KiB)

Creating archive: test.7z

Add new data to archive: 1 file, 12 bytes (1 KiB)

Files read from disk: 1
Archive size: 138 bytes (1 KiB)
Everything is Ok
5: exec 7za l test.7z

7-Zip (a) 21.07 (LE) : Copyright (c) 1999-2021 Igor Pavlov : 2021-12-26
 64-bit locale=C UTF8=- Threads:2

Scanning the drive for archives:
1 file, 138 bytes (1 KiB)

Listing archive: test.7z

--
Path = test.7z
Type = 7z
Physical Size = 138
Headers Size = 122
Method = LZMA2:12
Solid = -
Blocks = 1

   Date      Time    Attr         Size   Compressed  Name
------------------- ----- ------------ ------------  ------------------------
1970-01-01 08:00:00 .R..A           12           16  test.txt
------------------- ----- ------------ ------------  ------------------------
1970-01-01 08:00:00                 12           16  1 files
6: exec busybox rm test.txt
7: exec busybox ls
7za        busybox    ld         nm         readelf    strings    test.7Z
addr2line  c          ld.bfd     objcopy    readelf    strings    wzq
ar         elfedit    libc.so    objdump    size       strip
as         gprof      libc.so    ranlib     size       strip
8: exec 7za x -mmt=1 test.7z -aoa

7-Zip (a) 21.07 (LE) : Copyright (c) 1999-2021 Igor Pavlov : 2021-12-26
 64-bit locale=C UTF8=- Threads:2

Scanning the drive for archives:
1 file, 138 bytes (1 KiB)

Extracting archive: test.7z
--
Path = test.7z
Type = 7z
Physical Size = 138
Headers Size = 122
Method = LZMA2:12
```

```
Solid = -
Blocks = 1

Everything is Ok

Size:       12
Compressed: 138
9: exec busybox cat test.txt
hello world
10: exec readelf -h wzq
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x10670
  Start of program headers:          64 (bytes into file)
  Start of section headers:          86552 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         7
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
next:wzq
```

```
wzq Author:Gwins7
15|-------------|
14|             |
13|             |
12|             |
11|             |
10|             |
 9|             |
 8|      oo+    |
 7|      ox     |
 6|    oxxxxo   |
 5|     o       |
 4|             |
 3|             |
 2|             |
 1|-------------|
  ABCDEFGHIJKLMNO
Lastdown:j8
```

问题与解决方案

debug手段较为有限

与决赛第一阶段不同，本阶段中的应用程序均为较为复杂的应用程序，除了一开始提到的难以准确获取系统调用号的问题之外，在debug上手段也非常有限。一方面，用户态程序的debug较为困难，在工程开始时我们的程序均不含有symbol table，除了在gdb内打断点较为困难外，阅读没有标记的反汇编代码难以理解问题的所在，加之复合程序本身阅读源码非常困难，很多时候无法给出针对性的解决方法。在主办方提供的测试中，debug尚可以花费大量时间阅读源码与寻找错误点进行解决，而在移植应用程序阶段中，由于同样使用了较为复杂的实际项目源码编译而成的程序，debug也变成了一个几乎不可能的任务，如果遇到了用户态出现的handle_other，基本就没有办法解决了。幸运的是，由于我们本身实现内核较为完备，在我们选择的应用程序测试用例内没有出现上述问题，只在7za运行时存在内存溢出的问题，通过调大qemu内存也已经加以解决。

复杂机制引入的较为难以解决的问题

本阶段我们引入了一些较为复杂的机制，如上面提到的改进后的内存回收与swap机制，在这一阶段实现中，我们经常遇到一些难以理解的问题（尤其是附加上了随机性的问题），同时由于内核本身运行的连续性与持久性存储的复杂性，经常遇到如单独跑某个测试能够跑通而连续跑就卡住（有时调换顺序又能解决问题），或qemu上能跑，但一上板测试就卡住（甚至随机卡住），或难以理解的内存泄漏，或变量定义在头文件内就直接出现难以理解行为等一系列千奇百怪的问题。这个类型的问题通常都非常难以解决，这也是我们在整个编写与调试过程中难度最大的一个部分。

找到合适的应用程序

考虑到我们在第一阶段中所使用的是riscv64-musl库，在寻找应用程序时，我们一开始的寻找策略是在主办方提供的alpine linux package仓库内寻找合适的项目并想办法拿到源码进行编译，但是经过一段时间的尝试，这条路是难以走通的，主要原因是我们无法从仓库中直接获得可以编译的源码（因为package仓库类似于ubuntu的apt对应仓库，我们只能拿到打一些alpine linux可以用的apk安装包文件（内含路径和一些参数设置，但没有源码），而无法拿到可编译的源码）。同时，在我们自己实现的操作系统上，要选择符合我们条件的可以用于展示的程序也较为困难。最后，我们采用的选择方法是，在网上直接搜索项目的源码，并尝试在本地使用riscv64交叉编译链进行编译，但就算是这样，要找到能够使用riscv64交叉编译链（最好使用musl库）进行编译的程序源码也非常困难，最后终于选出了本次的三种应用程序，其中p7zip甚至只能使用glibc库编译，但是最后运行时也没有出现很大的问题，这也是比较幸运的。

k210本身的性能瓶颈

K210本身并不是一块性能较高的开发板，我们能够使用的CPU时钟频率是有限的，内存大小是有限的，SD卡的读写带宽亦是有限的。在这样的条件下，除了对资源的使用需要极度节省之外，为了最大程度提升性能（或者只是不至于上板时卡住或超时），需要在编写代码上多加考虑。在我们的代码中，除了对内存的利用效率极高之外，也实现了诸如fast_load_elf、bcache等用于优化性能的机制，以及为了适配ctx的需要而特化的extend pcb/fd机制与swap机制（但是由于板子本身的性能限制，最后仍然没有成功运行ctx 96，但在qemu上可以运行，这是较为遗憾的一点），这为我们的代码设计极大地增加了复杂性，也使得我们的系统设计更有价值。

总结

总的来说，在决赛第二阶段中，通过持之以恒的努力以及优化，我们构建出了一套功能较为完备的操作系统，并在系统中添加了不少我们自己独创的功能，最后能够成功运行三个官方提供的测试程序中的绝大部分测试用例且性能较为优秀，同时能够编译与运行我们自己选择的binutils、7za与wzq三个应用程序。本次设计中，较为可贵的一点是，与部分队伍选用较为成熟的操作系统内核（如rcore）作为基底不同，我们的操作系统除了框架结构外，几乎所有的代码均为自行编写实现。虽然仍然存在一些尚待解决的问题，但是最终实现的效果上是相当不错的，也取得了较为不错的成绩，较好地完成了本阶段的实现任务。