操作系统原理课程设计报告

姓　　名：　　杨明欣

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　计算机科学与技术 2106 班

学　　号：　　U202115514

指导教师：　　阳富民

分数	
教师签名	

2023 年 4 月 7 日

目　录

实验 challengeX 实现交互式 shell

1.1 实验目的

本实验旨在综合完成所有挑战实验并且最终实现交互式 shell：

1) 完成所有挑战实验
   a) 包括所有 challenge1；
   b) 特别是关键性挑战实验（所有 challenge3）；
2) 实现交互式 shell
   a) 类似 Bash 或其他 Linux Shell；
   b) 融合 PKE 之前所有的挑战实验的知识点；
   c) 需要有命令证明之前的挑战实验知识点已融合进去。

通过完成以上目标，最终能够在操作系统方面获得实际经验，提高对操作系统的整体理解水平。具体在本部分我将介绍 shell 的宏观实现方案，并在下面的部分详细介绍部分重要模块具体的实现逻辑。

1.2 实验内容

完成所有的挑战实验，如图 1.2-1 所示。

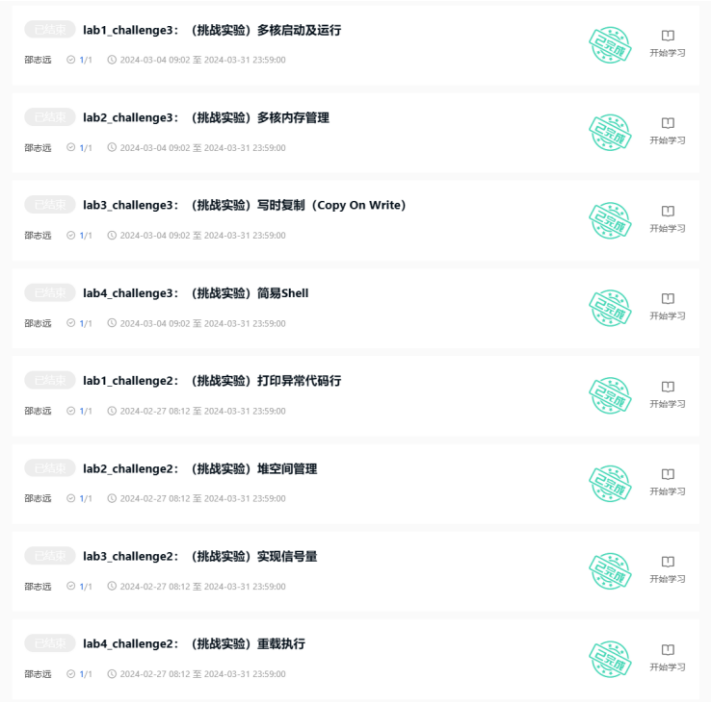

图 1.2-1 挑战实验提交截图

完成交互式 shell——融合其他挑战实验

将挑战实验融合进 shell 中，我选择通过继承 lab4_challenge3 简易 Shell，在它的基础上完成其他实验的融合。首先融合 lab1 中的两个挑战实验，两个实验因为是在虚拟内存管理实验之前完成的，因此尤其需要注意的就是虚拟地址和实地址之间的转换，转换正确的话实验内容上与之前的实验没有冲突；进一步需要融合 lab2 内存管理中的实验，lab2_challenge1 复杂缺页异常实验的实现较为简单，实验二堆空间管理则需要进一步将堆空间的管理方式进行修改，因此需要进行整体上的改动，在过程中需要细心；然后融合 lab3 进程管理中的实验，三个实验在融合时均无冲突，可以直接添加到实验中。

最终单核 shell 圆满完成。

完成交互式 shell——支持多核

对于多核的实现，需要明确对于唯一的资源，所有核会共享，所以需要控制并发。对于多份的资源，所有核应当独立占有，所以需要隔离资源。

具体细节会在下面进行展示，最终实验完成。

完成交互式 shell——实现交互化

需要完成交互，具体需要通过 scanfu 和 printfu 实现，具体实现在接下来会详细进行介绍，最终 shell 的代码实现如下：

```
1.  #include "user_lib.h"
2.  #include "string.h"
3.  #include "util/types.h"
4.
5.  #define Author "Mingxin_Yang"
6.  #define Host "ymx"
7.
8.  int main(int argc, char *argv[]) {
9.    printu("\n======== Shell Start ========\n\n");
10.
11.   printu("Author: %s\n", Author);
12.   printu("Operating System: RISC-V\n\n");
13.
14.   int start = 0;
15.   char str[20] = "cross";
16.   while(TRUE){
17.     char *command = (char*) better_malloc(100);
18.     char *para = (char*) better_malloc(100);
19.     // char *command = (char*) naive_malloc();
20.     // char *para = (char*) naive_malloc();
21.     char path[30];
22.     memset(path, '\0', strlen(path));
```

```
23.     read_cwd(path);
24.     printu("%s@%s:%s ", Author, Host, path);
25.     int a = scanfu("%s%s", command, para);
26.
27.     if(!strcmp(command, "exit")) break;
28.     printu("Next command: %s %s\n\n", command, para);
29.     printu("==========Command Start==========\n\n");
30.     int ret = 0;
31.     int pid = fork();
32.     if(pid == 0) {
33.       // printu("Next command: %s %s\n\n", command, para);
34.       ret = exec(command, para);
35.
36.       if (ret == -1)
37.       printu("exec failed!\n");
38.     }
39.     else{
40.       wait(pid);
41.       printu("==========Command End==========\n\n");
42.     }
43.   }
44.   exit(0);
45.   return 0;
46. }
```

1.3实现界面以及功能

实验运行命令（多核）：spike -p2 ./obj/riscv-pke /bin/app_shell /bin/app_alloc0

实验界面如图 1.3-1 所示。

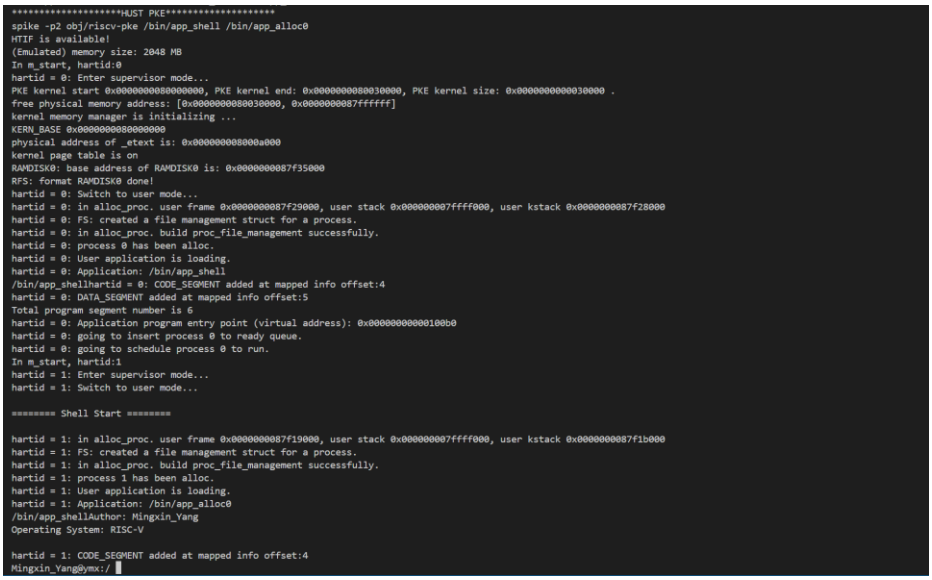

图 1.3-1 交互式 shell 运行界面

交互式 shell 支持的命令如下：

```
# 创建文件目录 sub_dir
/bin/app_mkdir /RAMDISK0/sub_dir
# 创建文件 ramfile1
/bin/app_touch /RAMDISK0/sub_dir/ramfile1
# 写入文件 ramfile1
/bin/app_echo /RAMDISK0/sub_dir/ramfile1
# 读出文件 ramfile1
/bin/app_cat /RAMDISK0/sub_dir/ramfile1
# 进入/RAMDISK0 文件目录
/bin/app_cd /RAMDISK0
# 支持相对路径打印当前目录下 sub_dir 文件目录下所有文件
/bin/app_ls ./sub_dir
# 打印/RAMDISK0 文件目录下所有文件
/bin/app_ls /RAMDISK0
# 显示历史命令
/bin/app_history
# 显示当前目录
/bin/app_pwd
# 验证写时复制实验
/bin/app_cow
# 验证信号量实验
/bin/app_semaphore
# 验证复杂缺页异常实验
/bin/app_sequence
# 验证打印异常代码行实验
/bin/app_errorline
# 验证打印用户程序调用栈实验
```

```
/bin/app_backtrace
# 验证堆空间管理实验
/bin/app_singlepageheap
# 支持相对路径进入当前目录下 sub_dir 文件目录
/bin/app_cd ./sub_dir
# 回退到上一文件目录
/bin/app_cd ..
# 退出 shell 系统
exit
```

1.4提交记录

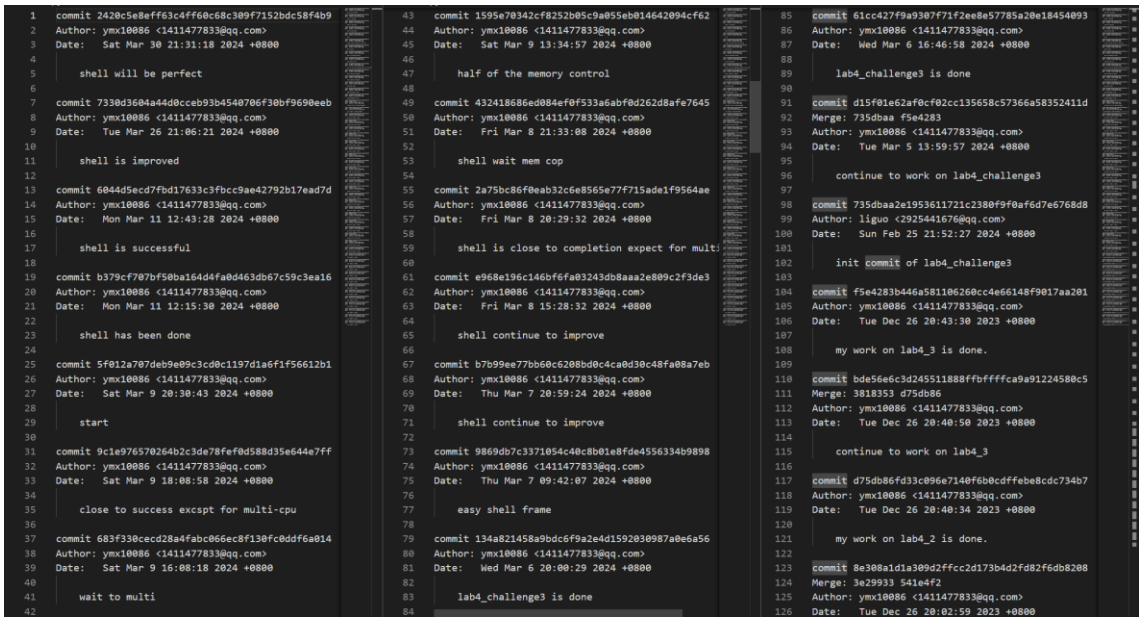

图 1.4-1 实验 git 提交记录

实验具体的提交记录如图 1.4-1 所示，可以看到 challengeX 实验差不多在一周内实现完成，难度还是比较大的。

1.5实验调试及课程建议

在完成实验 challengeX，实现交互式 shell 的过程中，我遇到了一些挑战，但也学到了很多关于操作系统和编程的知识。首先，我深入理解了 Shell 的工作原理，包括用户输入解析、命令执行等方面。其次，设计和实现交互函数也是关键，我花费了大量时间确保其能够准确接受屏幕的命令。进一步，我学会了如何处理系统调用和管理进程，这涉及到了 fork()、exec()等系统调用的使用。在编码过程中，我注重异常处理和错误检测，以确保 Shell 能够稳定运行，并给出有意义的错误提示。此外，我也尝试增加了一些额外功能，如历史命令记录，使得

Shell 更加实用和用户友好。最后，通过大量的测试和调试工作，我确保 Shell 在各种情况下都能够正常工作，提高了自己的编程能力和问题解决能力。这次实验让我受益匪浅，对我未来的学习和工作具有重要意义。

对于课程的建议主要就是在整个实验的过程中，我对于多核的实现逻辑并不清楚，对于系统内部的多核切换逻辑也始终抱有疑惑，在不断实验的过程中，我有些只能靠猜测或者是询问老师才能够得到解答，希望老师能够在之后的内容中进行补充，尤其是虚拟系统内部的多核实现逻辑，会帮助学生更好地完成实验。

难点一：挑战实验——堆空间

2.1 实验目的

本实验旨在对进程的虚拟地址空间进行管理，建议参考 Linux 的内存分配策略从而实现 malloc：

1) 通过修改 PKE 内核（包括 machine 文件下的代码）
2) 实现优化后的 malloc 函数
3) 使得应用程序两次申请块在同一页面

通过完成以上目标，最终能够在操作系统方面获得实际经验，提高对操作系统的整体理解水平。

2.2 实验内容

lab2_challenge2 堆空间管理

参考 Linux 的内存分配策略从而实现 malloc。因此我在实验中主要完成了以下任务：

1) 首先需要定义一个内存管理块 MCB 的数据结构，其中包括块大小以及下一个 MCB 块对应的地址，具体定义如下所示：

```
1.  // ! add for lab2_challenge2
2.  typedef struct mem_block {
3.  uint64 cap; // mem_block size
4.  uint64 next; //
5.  }mem_block;
```

2) 在进程中维护空闲内存块和占用内存块，修改 process 的数据结构以扩展对虚拟地址空间的管理，具体修改如下：

```
1.  typedef struct process_heap_manager
2.  {
3.     uint64 mem_used;
4.     uint64 mem_free;
5.     uint64 g_ufree_page;
6.
7.
8.     // points to the last free page in our simple heap.
```

```
    uint64 heap_top;
    // points to the bottom of our simple heap.
    uint64 heap_bottom;

    // the address of free pages in the heap
    uint64 free_pages_address[MAX_HEAP_PAGES];
    // the number of free pages in the heap
    uint32 free_pages_count;
} process_heap_manager;
```

3) 设计函数对进程的虚拟地址空间进行管理，借助以上内容具体实现 heap 扩展。对于进程中的内存块，我们实现了 insert_into_free 函数和 insert_into_used 函数进行管理，对于页内用过的 MCB 空间块调用 insert_into_used，对于未被使用的块则调用 insert_into_free，在进程申请空间时我们采用了教材中的首次适应算法，首先查看维护的空闲块队列中是否有空间大小的块，如果有，则将其从空闲队列中取出，进行数据赋值，并移到占用队列中；如果没有，再调用 alloc_page 申请新的页。具体的 insert_into_free 实现方法如下：

```
//
//// insert for free chain
//
static void insert_into_free(uint64 mem_va){
  uint64 hartid = read_tp();
 if (current[hartid]->user_heap.mem_free == -1){
  mem_block *mem_pa = (mem_block *)(user_va_to_pa((pagetable_t)current[hartid]->pagetable, (void *)mem_va));
  mem_pa->next = -1;
  current[hartid]->user_heap.mem_free = mem_va;
  return;
 }
 mem_block *mem_pa = (mem_block *)(user_va_to_pa((pagetable_t)current[hartid]->pagetable, (void *)mem_va));
 uint64 current_va = current[hartid]->user_heap.mem_free;
 mem_block *current_pa = (mem_block *)(user_va_to_pa((pagetable_t)current[hartid]->pagetable, (void *)current[hartid]->user_heap.mem_free));
 // judge if insert into the head of the chain
 if (current_pa->cap > mem_pa->cap){
  mem_pa->next = current[hartid]->user_heap.mem_free;
  current[hartid]->user_heap.mem_free = mem_va;
  return;
 }
 // insert into the correct location
```

```
22.  while (current_pa->next != -1){
23.   uint64 next_va = current_pa->next;
24.   mem_block *next_pa = (mem_block *)(user_va_to_pa((pagetable_t)current[hartid]->pa
      getable, (void *)next_va));
25.   if (next_pa->cap > mem_pa->cap) break;
26.   current_va = next_va;
27.   current_pa = next_pa;
28.  }
29.  mem_pa->next = current_pa->next;
30.  current_pa->next = mem_va;
31.  return;
32. }
```

4) 设计 malloc 函数和 free 函数对内存块进行管理。Malloc 函数首先查看是否空闲队列中有合适空间大小的块，判断是否需要申请空间，Free 函数则将空间放回空闲队列即可。具体 malloc 实现代码如下：

```
5)  void *better_alloc(uint64 n){
6)
7)   uint64 hartid = read_tp();
8)
9)   if (current[hartid]->user_heap.mem_free == -1) return alloc_from_used(n);
10)  uint64 va = current[hartid]->user_heap.mem_free;
11)  uint64 max_va = va;
12)  uint64 va_ptr;
13)  while (va != -1){
14)   max_va = va > max_va ? va : max_va;
15)   mem_block *pa = (mem_block *)(user_va_to_pa((pagetable_t)current[hartid]->pageta
      ble, (void *)va));
16)   if (pa->cap >= n + sizeof(mem_block)){
17)    va_ptr = va + sizeof(mem_block);
18)    alloc_from_free(va, n);
19)    return (void *)va_ptr;
20)   }
21)   va = pa->next;
22)  }
23)
24)  mem_block* max_pa = (mem_block*)user_va_to_pa((pagetable_t)current[hartid]->pag
      etable, (void *)max_va);
25)  if (max_va + max_pa->cap == current[hartid]->user_heap.g_ufree_page){
26)
27)   va_ptr = max_va + sizeof(mem_block);
28)   alloc_from_free_and_used(max_va, n);
29)   return (void *)va_ptr;
30)  }
```

```
31)  return alloc_from_used(n);
32) }
```

在实验过程中尤其需要注意内存的各种操作，因为内存管理本来就很复杂，细节比较多，需要完全理解内存映射的关系，同时尤其需要关注内存对齐，一旦出错需要一步步调试才能找到错误。

2.3 实验结果

最终实验结果为当两次申请的空间足够小时，会在同一页中，具体结果如图 2.3-1 所示。

```
Mingxin_Yang@ymx:/ /bin/app_singlepageheap
Next command: /bin/app_singlepageheap

==========Command Start============

hartid = 0: User call fork.
hartid = 0: will fork a child from parent 0.
hartid = 0: in alloc_proc. user frame 0x0000000087edb000, user stack 0x000000007ffff000, user kstack 0x0000000087ed4000
hartid = 0: FS: created a file management struct for a process.
hartid = 0: in alloc_proc. build proc_file_management successfully.
hartid = 0: process 4 has been alloc.
hartid = 0: do_fork map code segment at pa:0000000087f13000 of parent to child at va:0000000000010000.
hartid = 0: going to insert process 4 to ready queue.
hartid = 0: going to schedule process 4 to run.
hartid = 0: Application: /bin/app_singlepageheap
/bin/app_shellhartid = 0: CODE_SEGMENT added at mapped info offset:4
hartid = 0: DATA_SEGMENT added at mapped info offset:5
Total program segment number is 6
hartid = 0: Application program entry point (virtual address): 0x00000000000100b0
cross page
hartid = 0: User exit with code:0.
hartid = 0: going to insert process 0 to ready queue.
hartid = 0: going to schedule process 0 to run.
==========Command End============
```

图 2.3-1 堆空间管理

难点二：挑战实验——重载执行

3.1 实验目的

本实验旨在实现 exec 函数，在一个进程中启动另一个程序执行。其中可执行文件通过文件的路径名指定，从文件系统中读出。exec 会根据读入的可执行文件将原进程的数据段、代码段和堆栈段替换。

1) 通过修改 PKE 内核和系统调用，为用户程序提供 exec 函数的功能。
2) exec 接受一个可执行程序的路径名作为参数，表示要重新载入的 elf 文件。exec 函数在执行成功时不会返回，执行失败时返回-1。
3) 内核代码的修改可能包含添加系统调用、在内核中实现 exec 函数的功能。

通过完成以上目标，最终能够在操作系统方面获得实际经验，提高对操作系统的整体理解水平。

3.2 实验内容

当前进程清空

实现 exec 系统调用最简单的方法是直接覆盖当前进程，从而完成 exec 函数的功能。因此首先需要清空当前程序内的页，将其转化为有效页。然后类似于 alloc_process 重新进行进程创建。（也可以考虑直接创建一个新的进程进行替换，但是这样会浪费一部分空间）具体代码如下：

```
1.  // realease three level pagetable
2.  static void exec_clean_pagetable(pagetable_t page_dir) {
3.
4.      for (int i = 0; i < PGSIZE / sizeof(pte_t); i++) {
5.          pte_t* pte1 = page_dir + i;
6.          if (*pte1 & PTE_V) {
7.              pagetable_t page_mid_dir = (pagetable_t)PTE2PA(*pte1);
8.              for (int j = 0; j < PGSIZE / sizeof(pte_t); j++) {
9.                  pte_t* pte2 = page_mid_dir + j;
10.                 if (*pte2 & PTE_V) {
11.                     pagetable_t page_low_dir = (pagetable_t)PTE2PA(*pte2);
12.                     for (int k = 0; k < PGSIZE / sizeof(pte_t); k++) {
13.                         pte_t* pte3 = page_low_dir + k;
14.                         if (*pte3 & PTE_V) {
15.                             if (*pte3 & PTE_W & PTE_R) {
```

```
                    uint64 page = PTE2PA(*pte3);
                    free_page((void *)page);
                }
                (*pte3) &= ~PTE_V;
            }
        }
        free_page((void *)page_low_dir);
    }
    }
    free_page((void *)page_mid_dir);
  }
 }
 free_page((void *)page_dir);
}
```

目标程序加载

具体实现代码如下：

```
uint64 argv_va = current[hartid]->trapframe->regs.sp - arglen - 1;
argv_va = argv_va - argv_va % 8;
uint64 argv_pa = (uint64)user_va_to_pa(current[hartid]->pagetable, (void *)argv_va);
strcpy((char *)argv_pa, arg);

uint64 argvs_va = argv_va - 8;
uint64 argvs_pa = (uint64)user_va_to_pa(current[hartid]->pagetable, (void *)argvs_va);
*(uint64 *)argvs_pa = argv_va;

current[hartid]->trapframe->regs.a0 = 1;
current[hartid]->trapframe->regs.a1 = argvs_va;
current[hartid]->trapframe->regs.sp = argvs_va - argvs_va % 16;

load_bincode_from_host_elf(current[hartid], path);
```

通过查阅资料，获取参数的位置，将参数传递给当前的进程。使用 load_bincode_from_host_elf 函数加载进程的 elf 文件。

3.3 实验结果

可以成功运行 shell 中的所有命令。具体执行结果如图 3.3-1 所示。

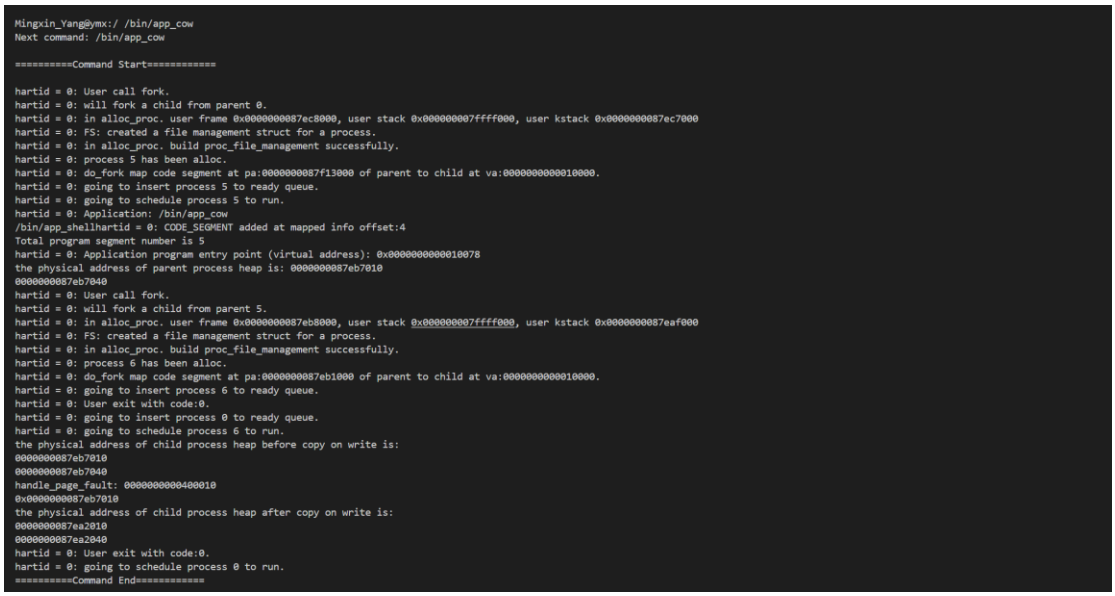

图 3.3-1 shell 命令运行

难点三：挑战实验融合过程

4.1 实验目的

本实验旨在将所有的挑战实验融合进入单核 shell 过程中，解决融合过程中出现的各种问题：

1) 由于挑战实验的实现时机许多实验功能没有完善，因此需要对于之前挑战实验的部分内容进行完善和修改
2) Lab4 的实验间没有冲突，因此挑战实验可以逐步合并前一个挑战实验，节省工作量

通过完成以上目标，最终能够在操作系统方面获得实际经验，提高对操作系统的整体理解水平。

4.2 实验内容

Lab4 简易 shell 实现

在完成 challengeX 实验的时候，比较方便入手的就是从 Lab4 简易 shell 实现的基础上进行完善，与此同时，lab4 中的三个挑战实验包括相对路径，重载执行以及简易 Shell 之间都完全没有冲突，因此可以逐步继承，完成 challengeX 的初步过程。

进一步融合其余挑战实验的时候，我选择从后向前融合，主要考虑到越靠后的实验，系统越完善，融合的方法越简单，需要修改的部分越少，更重要的是做的时间比较近，也比较熟悉。

Lab3 实验融合

在实现 lab3_challenge1 进程等待和数据段复制过程中，由于在之前完成挑战实验时没有完善的文件系统，因此在实现进程数据段复制的同时也需要完成文件系统的复制，包括文件的打开情况等，具体代码如下：

```
1.  child->pfiles->nfiles = parent->pfiles->nfiles;
2.   child->pfiles->cwd = parent->pfiles->cwd;
3.   for (int i = 0; i < MAX_FILES; i++) {
4.       child->pfiles->opened_files[i] = parent->pfiles->opened_files[i];
5.       if (child->pfiles->opened_files[i].f_dentry != NULL)
6.           child->pfiles->opened_files[i].f_dentry->d_ref++;
7.   }
8.
```

```
9.    child->status = READY;
10.   child->trapframe->regs.a0 = 0;
11.   child->parent = parent;
12.   child->trapframe->regs.tp = read_tp();
13.   insert_to_ready_queue(child);
```

Lab2 实验融合

融合 lab2_challenge1 复杂缺页异常非常简单，仅需要在缺页异常中补充好内容即可，lab2_challenge2 堆空间管理较为麻烦，因为之前进行堆空间管理的方式同样是以页为单位进行管理，因此需要重新完成，并更改堆空间（块为单位）管理的方式。

同时需要将所有的 naive_malloc 替换成 better_malloc，naïve_free 替换成 better_free。堆空间更改前后代码如下：

```
1.  case HEAP_SEGMENT:
2.
3.      // for (uint64 heap_block = current->user_heap.heap_bottom;
4.      //    heap_block < current->user_heap.heap_top; heap_block += PGSIZE) {
5.      //  uint64 parent_pa = lookup_pa(parent->pagetable, heap_block);
6.
7.      //  user_vm_map((pagetable_t)child->pagetable, heap_block, PGSIZE, parent_pa,
8.      //              prot_to_type(PROT_READ, 1));
9.
10.     //  pte_t *child_pte = page_walk(child->pagetable, heap_block, 0);
11.     //  *child_pte |= PTE_C;
12.     //  pte_t *parent_pte = page_walk(parent->pagetable, heap_block, 0);
13.     //  *parent_pte &= (~PTE_W);
14.     // }
15.     uint64 free_va = parent->user_heap.mem_free;
16.     while (free_va != -1){
17.         uint64 free_pa = (uint64)(user_va_to_pa((pagetable_t)parent->pagetable, (void *)f
    ree_va));
18.         cow_vm_map((pagetable_t)child->pagetable, free_va, free_pa);
19.         free_va = ((mem_block *)free_pa)->next;
20.     }
21.     uint64 used_va = parent->user_heap.mem_used;
22.     while (used_va != -1){
23.         uint64 used_pa = (uint64)(user_va_to_pa((pagetable_t)parent->pagetable, (void *)
    used_va));
24.         cow_vm_map((pagetable_t)child->pagetable, used_va, used_pa);
25.         used_va = ((mem_block *)used_pa)->next;
26.     }
27.
```

```
28.         // sprint("free_va : %d, used_va : %d\n", free_va, used_va);
29.         child->mapped_info[HEAP_SEGMENT].npages = parent->mapped_info[HEAP_SE
    GMENT].npages;
30.         memcpy((void*)&child->user_heap, (void*)&parent->user_heap, sizeof(parent->use
    r_heap));
31.         break;
```

Lab1 实验融合

需要注意的关键点为在完成 lab1 的挑战实验时，所有的虚拟内存空间管理还没有完善，因此需要通过下面的代码将对应的虚拟地址转化成物理地址，即可完成实验。

```
1.  char* pa = (char*)user_va_to_pa((pagetable_t)(current[hartid]->pagetable), (void*)buf);
```

4.3 实验结果

可以成功运行挑战实验的所有内容，以 lab3_challenge2 实现信号量为例，执行/bin/app_semaphore，具体结果如下图 4.3-1 所示。

```
hartid = 0: Will fork a child from parent 9.
hartid = 0: in alloc_proc. user frame 0x0000000087e36000, user stack 0x000000007ffff000, user kstack 0x0000000087e35000
hartid = 0: FS: created a file management struct for a process.
hartid = 0: in alloc_proc. build proc_file_management successfully.
hartid = 0: process 10 has been alloc.
hartid = 0: do_fork map code segment at pa:0000000087e42000 of parent to child at va:0000000000010000.
hartid = 0: going to insert process 10 to ready queue.
Child0 print 0
hartid = 0: going to schedule process 10 to run.
Child1 print 0
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 1
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 1
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 1
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 2
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 2
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 2
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 3
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 3
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 3
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 4
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 4
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 4
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 5
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 5
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 5
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 6
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 6
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 6
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 7
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 7
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 7
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 8
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 8
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: going to schedule process 10 to run.
Child1 print 8
hartid = 0: going to insert process 8 to ready queue.
hartid = 0: going to schedule process 8 to run.
Parent print 9
hartid = 0: going to insert process 9 to ready queue.
hartid = 0: User exit with code:0.
hartid = 0: going to insert process 0 to ready queue.
hartid = 0: going to schedule process 9 to run.
Child0 print 9
hartid = 0: going to insert process 10 to ready queue.
hartid = 0: User exit with code:0.
hartid = 0: going to schedule process 0 to run.
==========Command End===========
```

图 4.3-1 信号量测试

难点四：多核机制实现

5.1 实验目的

本实验旨在对于已经完善好的单核 shell 补充双核机制，使得其在实现 shell 程序的同时能够执行另一个程序：

1) 深入理解 lab1 和 lab2 中多核机制的实现
2) 对于 lab3 和 lab4 补充的进程和文件系统也能够实现支持
3) 对于进程 fork 过程中的各个资源能够实现互斥使用

通过完成以上目标，最终能够在操作系统方面获得实际经验，提高对操作系统的整体理解水平。

5.2实验内容

资源同步初始化

对于 challengeX 来说，在初始化多核中的文件系统，内存空间，系统等只需要初始化一次，在完成初始化后同样需要同步进程，保证系统顺利运行。

在 minit.c 中体现如下：

```
1.    if(mutex == 0) {
2.      mutex = 1;
3.      spike_file_init();
4.
5.      // init HTIF (Host-Target InterFace) and memory by using the Device Table Blob (DTB
      )
6.      // init_dtb() is defined above.
7.      init_dtb(dtb);
8.    }
9.    // ! add for lab1_challenge3
10.   sync_barrier(&htif_initialized, NCPU);
```

在 kernel 中体现如下：

```
1.    if(start_init == 0){
2.      start_init = 1;
3.      // init phisical memory manager
4.      pmm_init();
5.
```

```
    // build the kernel page table
    kern_vm_init();

    // now, switch to paging mode by turning on paging (SV39)
    enable_paging();
    // the code now formally works in paging mode, meaning the page table is now in use.
    sprint("kernel page table is on \n");

    // added @lab3_1
    init_proc_pool();

    // init file system, added @lab4_1
    fs_init();
}
```

加载对应程序

加载过程中不同的核要加载不同的程序。可以通过修改命令行参数的传递过程实现不同进程的加载，默认由 CPU0 加载第一个参数对应的程序，由 CPU1 加载第二个参数对应的程序，具体通过 parse_args 函数进行接受，命令行参数被读入 arg_bug_msg.argv 数组中，通过 read_tp()确定当前核需要的命令行参数 arg_bug_msg.argv[read_tp()]，进而得到需加载的 elf 文件的名称与位置。

资源扩展

在执行多核程序的时候，许多维护的资源或者空间需要变成多份，以支持不同的核并行地执行对应的程序，具体需要每个核单独维护的资源如下：

```
riscv_regs g_itrframe[NCPU];
process *current[NCPU] = {NULL};
process* ready_queue_head[NCPU] = {NULL};
```

资源互斥访问

在多核系统中，由于资源有限，当多个核心同时竞争有限资源时，需要使用互斥锁来确保资源的合理分配和程序的正确执行。互斥锁通过引入#include "spike_interface/atomic.h" 中的 spinlock_t 来实现，具体来说通过调用 spinlock_lock()函数进行加锁，通过 spinlock_unlock()进行解锁，具体需要互斥访问的函数如下 ：

```
process *alloc_process();
pte_t *page_walk(pagetable_t page_dir, uint64 va, int alloc);
void insert_to_ready_queue( process* proc ) ;
void free_page(void *pa) ;
void *alloc_page(void);
```

以 insert_to_ready_queue 函数为例，具体互斥访问的实现方式如下：

```
1.  //
2.  // insert a process, proc, into the END of ready queue.
3.  //
4.  spinlock_t insert_lock;
5.
6.  void insert_to_ready_queue( process* proc ) {
7.    spinlock_lock(&insert_lock);
8.    uint64 hartid = read_tp();
9.    sprint("hartid = %lld: going to insert process %d to ready queue.\n", hartid, proc->pid);
10.   // if the queue is empty in the beginning
11.   if( ready_queue_head[hartid] == NULL ){
12.     proc->status = READY;
13.     proc->queue_next = NULL;
14.     ready_queue_head[hartid] = proc;
15.     spinlock_unlock(&insert_lock);
16.     return;
17.   }
```

进程转换

需要注意程序在进入用户态的时候再次返回时需要根据当前进程中寄存器的值选择返回的对应进程，因此在实现过程中需要尤其注意 CPU 核编号的传递。具体实现代码如下：

```
1.  // init proc[i]'s vm space
2.  procs[i].trapframe = (trapframe *)alloc_page(); // trapframe, used to save context
3.  memset(procs[i].trapframe, 0, sizeof(trapframe));
4.
5.  // page directory
6.  procs[i].pagetable = (pagetable_t)alloc_page();
7.  memset((void *)procs[i].pagetable, 0, PGSIZE);
8.
9.  procs[i].kstack = (uint64)alloc_page() + PGSIZE; // user kernel stack top
10. uint64 user_stack = (uint64)alloc_page();   // phisical address of user stack bottom
11. procs[i].trapframe->regs.sp = USER_STACK_TOP;  // virtual address of user stack top
12.  procs[i].trapframe->regs.tp = hartid; // 十分重要
```

5.3 实验结果

支持多核同时并行执行程序，如下图 5.3-1 所示为执行命令 spike -p2 ./obj/riscv-pke /bin/app_shell /bin/app_alloc0，可以看到其同时执行了两个程

序。

```
******************HUST PKE******************
spike -p2 obj/riscv-pke /bin/app_shell /bin/app_alloc0
HTIF is available!
(Emulated) memory size: 2048 MB
In m_start, hartid:0
hartid = 0: Enter supervisor mode...
PKE kernel start 0x0000000080000000, PKE kernel end: 0x0000000080030000, PKE kernel size: 0x0000000000030000 .
free physical memory address: [0x0000000080030000, 0x000000087fffffff]
kernel memory manager is initializing ...
KERN_BASE 0x0000000080000000
physical address of _etext is: 0x000000008000a000
kernel page table is on
RAMDISK0: base address of RAMDISK0 is: 0x0000000087f35000
RFS: format RAMDISK0 done!
hartid = 0: Switch to user mode...
hartid = 0: in alloc_proc. user frame 0x0000000087f29000, user stack 0x000000007ffff000, user kstack 0x0000000087f28000
hartid = 0: FS: created a file management struct for a process.
hartid = 0: in alloc_proc. build proc_file_management successfully.
hartid = 0: process 0 has been alloc.
hartid = 0: User application is loading.
hartid = 0: Application: /bin/app_shell
/bin/app_shellhartid = 0: CODE_SEGMENT added at mapped info offset:4
hartid = 0: DATA_SEGMENT added at mapped info offset:5
Total program segment number is 6
hartid = 0: Application program entry point (virtual address): 0x00000000000100b0
hartid = 0: going to insert process 0 to ready queue.
hartid = 0: going to schedule process 0 to run.
In m_start, hartid:1
hartid = 1: Enter supervisor mode...
hartid = 1: Switch to user mode...

======= Shell Start =======

hartid = 1: in alloc_proc. user frame 0x0000000087f19000, user stack 0x000000007ffff000, user kstack 0x0000000087f1b000
hartid = 1: FS: created a file management struct for a process.
hartid = 1: in alloc_proc. build proc_file_management successfully.
hartid = 1: process 1 has been alloc.
hartid = 1: User application is loading.
hartid = 1: Application: /bin/app_alloc0
/bin/app_shellAuthor: Mingxin_Yang
Operating System: RISC-V

hartid = 1: CODE_SEGMENT added at mapped info offset:4
Mingxin_Yang@ymx:/ /bin/app_cow
Next command: /bin/app_cow

=========Command Start===========

hartid = 0: User call fork.
hartid = 0: will fork a child from parent 0.
Total program segment number is 5
hartid = 1: Application program entry point (virtual address): 0x0000000000010078
hartid = 1: going to insert process 1 to ready queue.
hartid = 1: going to schedule process 1 to run.
hartid = 0: in alloc_proc. user frame 0x0000000087efc000, user stack 0x000000007ffff000, user kstack 0x0000000087efb000
hartid = 0: FS: created a file management struct for a process.
hartid = 0: in alloc_proc. build proc_file_management successfully.
hartid = 0: process 2 has been alloc.
the physical address of parent process heap is: hartid = 0: do_fork map code segment at pa:0000000087f13000 of parent to child at va:0000000000010000.
0000000087f07010
hartid = 0: going to insert process 2 to ready queue.
0000000087f07040
hartid = 0: going to schedule process 2 to run.
hartid = 1: User call fork.
hartid = 1: will fork a child from parent 1.
hartid = 1: in alloc_proc. user frame 0x0000000087ef2000, user stack 0x000000007ffff000, user kstack 0x0000000087f04000
hartid = 1: FS: created a file management struct for a process.
hartid = 1: in alloc_proc. build proc_file_management successfully.
hartid = 1: process 3 has been alloc.
hartid = 0: Application: /bin/app_cow
```

图 5.3-1 多核执行程序

难点五：实现交互性

6.1实验目的

本实验旨在对于已经完善好的 shell 程序，使得其支持交互运行命令

1) 实现 scanfu 函数从屏幕中读入字符串
2) 利用 printfu 和 scanfu 实现 shell 的交互

通过完成以上目标，最终能够在操作系统方面获得实际经验，提高对操作系统的整体理解水平。

6.2实验内容

完成交互式 shell——scanfu 实现：

完成交互式 shell 首先要完善的功能就是实现屏幕文本读入功能，通过 scanfu 和 printfu 才能实现交互。首先参考了 vsnprintf 对于格式化字符串和参数的处理程序，独立实现了对于 vsnscanf，使得其支持多个字符串和整型变量的输入（对于交互式 shell 来说可以满足所有的要求），具体实现代码如下：

```
1.  int32 vsnscanf(const char *in, const char *format, va_list vl)
2.  {
3.   const char *p = format;
4.   bool isFormat = FALSE;
5.   int32 counts = 0;
6.    int32 argcounts = 0;
7.
8.   for (; *p; p++){
9.    if (isFormat){
10.    switch (*p){
11.    case 's':
12.        argcounts ++;
13.     char *res = va_arg(vl, char *);
14.        while(*in == ' ') in++;
15.        if (*in != '\n' && *in != '\0' && *in != ' ') counts++;
16.    while (*in != '\n' && *in != '\0' && *in != ' '){
17.     *res++ = *in++;
18.    }
19.        if (*in == '\n' && *in == '\0') break;
20.    *res = '\0';
```

```
21.         isFormat = FALSE;
22.       break;
23.     default:
24.       break;
25.     }
26.   }
27.   else if (*p == '%')
28.     isFormat = TRUE;
29.   }
30.   if (argcounts != counts) return 0;
31.   return argcounts;
32. }
```

进一步通过在后端调用 spike_file_read(stderr, pa, 256);即可完成对于屏幕信息的读入。

具体实现 scanfu 的函数调用代码如下：

```
1.  int scanfu(const char* s, ...) {
2.    va_list vl;
3.    va_start(vl, s);
4.
5.    char in[256];  // fixed buffer size.
6.
7.    // make a syscall to implement the required functionality.
8.    do_user_call(SYS_user_scanf, (uint64)in, 0, 0, 0, 0, 0, 0);
9.
10.   int res = vsnscanf(in, s, vl);
11.
12.   va_end(vl);
13.
14.   return res;
15. }
```

6.3 实验结果

通过 scanfu 函数可以正确输入包括仅有一个输入或者多个输入。输入仅单个函数时如图 6.3-1 所示。可以看到其加载程序并且顺利执行。

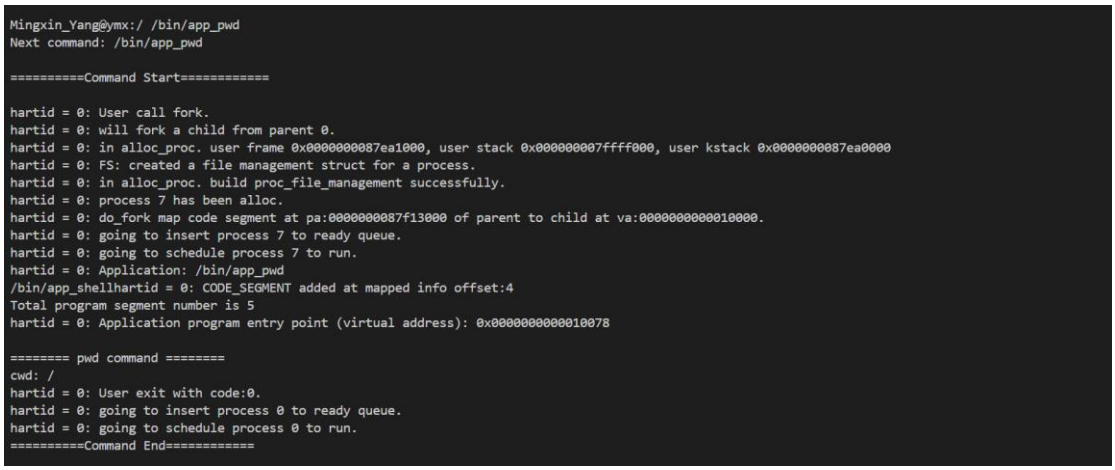

图 6.3-1 输入为单个函数

输入为函数及其参数时如图 6.3-2 所示。可以看到其加载程序并且顺利执行。

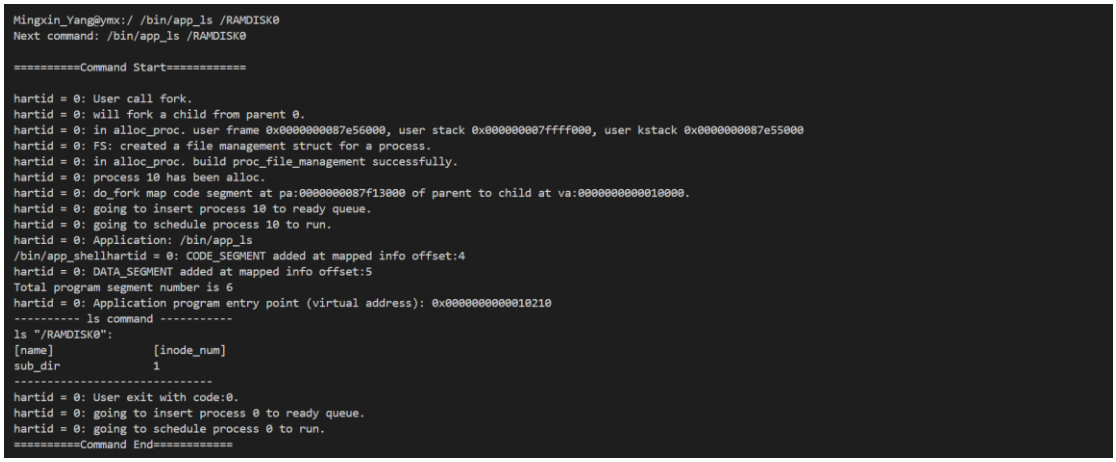

图 6.3-2 输入为函数及其参数

加分项：问题解决

- **是否能（通过 ls 命令）列出 hostfs 文件系统里面的文件？**

答：通过 ls 命令列出 hostfs 文件系统中的文件。hostfs 是一个用户空间文件系统，通常用于将主机的文件系统挂载到虚拟机中。但是在系统中具体执行 ls 命令时会发现 hostfs 文件系统并没有实现，因此需要进一步完善 hostfs 文件系统才可以查看文件。

- **压力测试：写个 app，一个进程 fork 出 100 个子进程，每个子进程 malloc 出一个长度为 1KB 的数组（长度可调整），往里面赋值，看他跨不跨；或者，分析为啥不能跑更大规模的应用，以上测试中进程创建的上限在哪？**

答：测试方案：

1) 编写一个应用程序，该程序创建一个主进程，并在主进程中使用 fork() 函数创建 100 个子进程。
2) 每个子进程使用 malloc()函数分配一个长度为 1KB（或其他可调整的长度）的数组，并将数组中的元素初始化。
3) 测试程序运行时，监测系统的资源使用情况，包括内存、CPU 等。

分析：

1) 系统资源限制： 在进行测试之前，需要清楚系统的资源限制，包括可用内存大小、最大进程数等。
2) 内存使用： 每个子进程 malloc 出 1KB 的数组，如果 100 个子进程同时存在，总共会占用 100KB 的内存。如果系统内存足够，应该不会出现问题。但是，如果内存不足，就可能导致内存不足的错误。
3) 进程数限制： 每个系统都有最大进程数的限制。如果创建的子进程数量超过了系统的最大进程数限制，就会导致创建失败。

测试结果：

1) 如果系统资源足够，程序可以顺利运行，并且每个子进程都能够成功创建并分配内存。
2) 如果系统资源有限，可能会出现内存不足或者进程创建失败的情况，导致程序无法正常运行。

总结：由于本系统的内存使用充足，但是进程数限制，资源限制，导致创建的进程数量也有限。

- **其他加分项，比如通过后台执行（&）同时运行多个进程；管道功能（|），一个进程的输出作为另一个进程的输入；其他功能，如命令历史、环境变量**

等？

答：历史变量显示：

可以通过执行/bin/app_history 显示历史命令，具体执行结果如图 6.3-1 所示。

```
Mingxin_Yang@ymx:/ /bin/app_history
Next command: /bin/app_history

==========Command Start============

hartid = 0: User call fork.
hartid = 0: will fork a child from parent 0.
hartid = 0: in alloc_proc. user frame 0x0000000087e43000, user stack 0x000000007ffff000, user kstack 0x0000000087e39000
hartid = 0: FS: created a file management struct for a process.
hartid = 0: in alloc_proc. build proc_file_management successfully.
hartid = 0: process 11 has been alloc.
hartid = 0: do_fork map code segment at pa:0000000087f13000 of parent to child at va:0000000000010000.
hartid = 0: going to insert process 11 to ready queue.
hartid = 0: going to schedule process 11 to run.
hartid = 0: Application: /bin/app_history
/bin/app_shellhartid = 0: CODE_SEGMENT added at mapped info offset:4
Total program segment number is 5
hartid = 0: Application program entry point (virtual address): 0x0000000000010078

======== history command ========
hartid = 0: User call history.
/bin/app_shell
/bin/app_alloc0
/bin/app_cow
/bin/app_cow
/bin/app_pwd
/bin/app_ls
/bin/app_mkdir
/bin/app_ls
/bin/app_history
hartid = 0: User exit with code:0.
hartid = 0: going to insert process 0 to ready queue.
hartid = 0: going to schedule process 0 to run.
==========Command End============
```

图 6.3-1 app_history

完善的系统文件展示：

可以通过执行/bin/app_ls 完善地展示文件目录下的所有文件，递归展示，具体展示方式为首先执行下面的多个文件或文件目录的创建命令：

1. /bin/app_mkdir /RAMDISK0/sub_dir1
2. /bin/app_mkdir /RAMDISK0/sub_dir1/sub_dira
3. /bin/app_touch /RAMDISK0/sub_dir1/ramfile1
4. /bin/app_mkdir /RAMDISK0/sub_dir2

接下来执行/bin/app_ls /RAMDISK0/命令，则可以展示所有的文件，具体结果如图 6.3-2 所示。

```
Mingxin_Yang@ymx:/ /bin/app_ls /RAMDISK0
Next command: /bin/app_ls /RAMDISK0

==========Command Start============

hartid = 0: User call fork.
hartid = 0: will fork a child from parent 0.
hartid = 0: in alloc_proc. user frame 0x0000000087e82000, user stack 0x000000007ffff000, user kstack 0x0000000087e81000
hartid = 0: FS: created a file management struct for a process.
hartid = 0: in alloc_proc. build proc_file_management successfully.
hartid = 0: process 7 has been alloc.
hartid = 0: do_fork map code segment at pa:0000000087f13000 of parent to child at va:0000000000010000.
hartid = 0: going to insert process 7 to ready queue.
hartid = 0: going to schedule process 7 to run.
hartid = 0: Application: /bin/app_ls
/bin/app_shellhartid = 0: CODE_SEGMENT added at mapped info offset:4
hartid = 0: DATA_SEGMENT added at mapped info offset:5
Total program segment number is 6
hartid = 0: Application program entry point (virtual address): 0x0000000000010210
---------- ls command ----------
ls "/RAMDISK0":
[name]              [inode_num]
sub_dir1            1
   ---sub_dira              2
   ---ramfile1              3
sub_dir2            4
------------------------------
hartid = 0: User exit with code:0.
hartid = 0: going to insert process 0 to ready queue.
hartid = 0: going to schedule process 0 to run.
==========Command End============
```

图 6.3-2 执行/bin/app_ls /RAMDISK0/命令