華中科技大學

操作系统原理课程实验报告

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 1 月 1 日

目　录

实验一　系统调用、异常和外部中断 1

1.1　实验目的 1
1.2　实验内容 1
1.3　实验调试及心得 4

实验二　内存管理 5

1.1　实验目的 5
1.2　实验内容 5
1.3　实验调试及心得 8

实验三　进程管理 9

1.1　实验目的 9
1.2　实验内容 9
1.3　实验调试及心得 11

挑战实验　进程等待和数据段复制 13

1.1　实验目的 13
1.2　实验内容 13
1.3　实验调试及心得 15

实验一 系统调用、异常和外部中断

1.1 实验目的

本实验旨在深入了解操作系统的基本机制，特别是系统调用、异常和外部中断。通过进行相关实验，我们将能够实现以下目标：

1) 理解系统调用的概念及其在操作系统中的实现。
2) 掌握异常处理的原理，包括异常的产生、传递和处理机制。
3) 研究中断的机制，了解程序如何触发操作系统的响应。

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

1.2 实验内容

lab1_1 系统调用：

lab1_1 实验在了解和掌握操作系统中系统调用机制的实现原理，进行完整的操作系统代码梳理后，具体思路如下：

1) app 的系统调用：从应用程序出发，发现 user/app_helloworld.c 文件中有两个函数调用：printu 和 exit。通过代码跟踪，我们发现这两个函数最终在 user/user_lib.c 中实现，而且它们最终都转换为对 do_user_call 函数的调用。
2) 系统调用的触发时机：深入研究 do_user_call 函数，发现它通过 ecall 指令来完成系统调用。在执行 ecall 指令之前，所有参数（do_user_call 函数的 8 个参数）已经被加载到 RISC-V 机器的 a0 到 a7 寄存器中，这一步是编译器生成的代码自动完成的。
3) 系统调用和内核堆栈的处理：总的来说就是保存进程现场，切换到内核栈，执行 S 模式下的 trap 入口函数，如果系统调用是由用户程序触发的，即 read_csr(scause)==CAUSE_USER_ECALL 则执行 handle_syscall()函数。最后，调用 switch_to()函数返回当前进程。
4) 完成 lab1_1：将 panic 语句替换为对 do_syscall()函数的调用，这样我们就能够通过系统调用完成特定的任务，而不会触发 panic。

最终实现的代码如下：

```
1. //
2. // handling the syscalls. will call do_syscall() defined in kernel/s
   yscall.c
3. //
4. static void handle_syscall(trapframe *tf) {
5.   // tf->epc points to the address that our computer will jump to af
   ter the trap handling.
6.   // for a syscall, we should return to the NEXT instruction after i
   ts handling.
7.   // in RV64G, each instruction occupies exactly 32 bits (i.e., 4 By
   tes)
8.   tf->epc += 4;
9.
10.  // TODO (lab1_1): remove the panic call below, and call do_syscall
    (defined in
11.  // kernel/syscall.c) to conduct real operations of the kernel side
    for a syscall.
12.  // IMPORTANT: return value should be returned to user app, or else
   , you will encounter
13.  // problems in later experiments!
14.  //panic( "call do_syscall to accomplish the syscall and lab1_1 her
   e.\n" );
15.  do_syscall(tf->regs.a0, tf->regs.a1, tf->regs.a2, tf->regs.a3, tf-
   >regs.a4, tf->regs.a5, tf->regs.a6, tf->regs.a7);
16.}
```

lab1_2 异常处理

lab1_2 异常处理的 app 中，（在用户 U 模式下执行的）应用企图执行 RISC-V 的特权指令 csrw sscratch, 0。该指令会修改 S 模式的栈指针，如果允许该指令的执行，执行的结果可能会导致系统崩溃。因此，需要通过调用 handle_illegal_instruction 函数完成异常指令处理，阻止 app_illegal_instruction 的执行。具体思路如下：

1) app 的非法指令：从应用程序的角度出发，我们观察到 user/app_illegal_instruction.c 文件中的代码尝试执行一个在用户模式（U 模式）下不允许运行的特权级指令：csrw sscratch, 0。
2) 非法指令触发异常：这个“异常事件”被识别为非法指令异常，对应的异常码是 CAUSE_ILLEGAL_INSTRUCTION，其值为 02。
3) 异常交付：对于 CAUSE_ILLEGAL_INSTRUCTION 这类异常，我们通过代码追踪到 m_start()函数执行 delegate_traps()函数时的代理规则。查看相关代码发现，这类异常并没有交给 S 模式处理，而是由 M 模式处理。

4) Trap 入口：寻找 M 模式的 trap 处理入口，我们在 kernel/machine/mtrap_vector.S 文件中找到了 mtrapvec 汇编函数。在该函数中，调用了 handle_mtrap()函数。

5) 完成 lab1_2：将 handle_mtrap()函数中对异常的处理逻辑从 panic 替换为调用 handle_illegal_instruction()函数。

最终实现的代码如下：

```
1.     case CAUSE_ILLEGAL_INSTRUCTION:
2.       // TODO (lab1_2): call handle_illegal_instruction to implement
   illegal instruction
3.       // interception, and finish lab1_2.
4.       //panic( "call handle_illegal_instruction to accomplish illega
   l instruction interception for lab1_2.\n" );
5.       handle_illegal_instruction();
6.       break;
```

lab1_3 （外部）中断

lab1_3 （外部）中断完成 PKE 操作系统内核未完成的时钟中断处理过程，使得它能够完整地处理时钟中断。具体思路如下：

1) 时钟中断的初始化和触发：在 m_start 函数中新增了 timerinit()函数，该函数设置下一次时钟中断触发的时间，并启用 M 模式中的时钟中断（MIE_MTIE 位）。时钟中断的触发与 lab2_1 相似。不赘述。

2) 时钟中断处理：在 S 模式的 smode_trap_handler 函数中，检查 scause 寄存器的值，如果是由 M 模式传递上来的时钟中断动作（CAUSE_MTIMER_S_TRAP），则调用 handle_mtimer_trap() 函数进行处理。

3) 完成 lab1_3：在 handle_mtimer_trap()函数中，首先对全局变量 g_ticks 进行加一操作，用于记录时钟中断的次数。其次，清零 SIP_SSIP 位，以确保下次时钟中断时，M 模式能够再次触发 S 模式的中断处理。这样，时钟中断的处理流程就得以完成。

具体代码的实现如下：

```
1. //
2. // added @lab1_3
3. //
4. void handle_mtimer_trap() {
5.   sprint("Ticks %d\n", g_ticks);
6.   // TODO (lab1_3): increase g_ticks to record this "tick", and t
   hen clear the "SIP"
```

```
    // field in sip register.
    // hint: use write_csr to disable the SIP_SSIP bit in sip.

    //panic( "lab1_3: increase g_ticks by one, and clear SIP field in sip register.\n" );
    g_ticks++;
    write_csr(sip, 0);

}
```

1.3 实验调试及心得

在完成这三个操作系统实验的过程中，我深入理解了操作系统底层机制。通过分析应用程序和内核代码，我逐步追踪和理解了系统调用的执行流程，异常的触发机制以及时钟中断在 M 模式和 S 模式之间的处理流程。实践中，我自主完成了系统调用的实现，修改了异常处理逻辑，用实际的代码实现了对非法指令异常的处理。

这些实验不仅增强了我对操作系统原理的理解，还培养了我分析和解决底层问题的能力，为今后深入研究和应用操作系统奠定了坚实基础。

实验二 内存管理

1.1 实验目的

本实验旨在使学生深入理解操作系统中的内存管理机制。通过进行相关实验，我们将能够实现以下目标：

1) 掌握页表的结构和工作原理。
2) 理解页表的实现，实践虚拟内存管理。
3) 处理页错误异常。

通过完成以上目标，将在实际的操作系统代码实现中，深入了解内存管理的关键概念和实践技巧，为更深入地理解操作系统底层原理奠定基础。

1.2 实验内容

lab2_1 虚实地址转换

lab2_1 虚实地址转换要求实现 user_va_to_pa()函数，完成给定逻辑地址到物理地址的转换。具体思路如下：

1) 理解用户空间的系统调用路径：基本与 lab1_1 的调用过程相同。
2) 用户空间到物理地址的转换：在 sys_user_print 函数中，需要将用户空间的 buf 地址转换为物理地址，以便后续在内核空间中操作。转换操作依赖于 user_va_to_pa 函数，该函数位于 vmm.c 文件中。
3) lab2_1 完成：在 user_va_to_pa 函数中完成虚拟地址到物理地址的转换，调用 page_walk 找到 va 所对应的页表项，然后进行转换映射到物理地址，则完成实验。

具体代码如下：

```
1. /* --- user page table part --- */
2. //
3. // convert and return the corresponding physical address of a virtua
   l address (va) of
4. // application.
5. //
6. void *user_va_to_pa(pagetable_t page_dir, void *va) {
7.   // TODO (lab2_1): implement user_va_to_pa to convert a given user
   virtual address "va"
```

```
  // to its corresponding physical address, i.e., "pa". To do it, we need to walk
  // through the page table, starting from its directory "page_dir", to locate the PTE
  // that maps "va". If found, returns the "pa" by using:
  // pa = PYHS_ADDR(PTE) + (va & (1<<PGSHIFT -1))
  // Here, PYHS_ADDR() means retrieving the starting address (4KB aligned), and
  // (va & (1<<PGSHIFT -1)) means computing the offset of "va" inside its page.
  // Also, it is possible that "va" is not mapped at all. in such case, we can find
  // invalid PTE, and should return NULL.
  // panic( "You have to implement user_va_to_pa (convert user va to pa) to print messages in lab2_1.\n" );

  pte_t *pte_va = page_walk(page_dir, (uint64) va, 0);
  uint64 pa = 0;
  if (pte_va)
    pa = PTE2PA(*pte_va) + (((uint64) va) & ((1 << PGSHIFT) - 1));
  return (void *) pa;

}
```

lab2_2 简单内存分配和回收

lab2_2 简单内存分配和回收需要完成 naive_free 对应的功能，并获得预期的结果输出。具体思路如下：

1) 内存分配：sys_user_allocate_page 函数分配物理页面 pa，并将其映射到 g_ufree_page 对应的逻辑地址 va。g_ufree_page 在 process.c 文件中的定义表明，动态分配的内存是从 USER_FREE_ADDRESS_START 开始的。
2) 内存回收：回收一个给定 va 对应的页表项（PTE）是内存回收的第一步。如果找到 PTE，可以通过其内容得知 va 对应物理页的首地址 pa。接下来，需要回收 pa 对应的物理页，并将 PTE 中的 Valid 位置为 0。

具体实现的代码如下：

```
//
// unmap virtual address [va, va+size] from the user app.
// reclaim the physical pages if free!=0
//
void user_vm_unmap(pagetable_t page_dir, uint64 va, uint64 size, int free) {
```

```
  // TODO (lab2_2): implement user_vm_unmap to disable the mapping of the virtual pages
  // in [va, va+size], and free the corresponding physical pages used by the virtual
  // addresses when if 'free' (the last parameter) is not zero.
  // basic idea here is to first locate the PTEs of the virtual pages, and then reclaim
  // (use free_page() defined in pmm.c) the physical pages. lastly, invalidate the PTEs.
  // as naive_free reclaims only one page at a time, you only need to consider one page
  // to make user/app_naive_malloc to behave correctly.
  //panic( "You have to implement user_vm_unmap to free pages using naive_free in lab2_2.\n" );

  pte_t *pte_containing_va = page_walk(page_dir, va, 0);
  if (pte_containing_va) {
    free_page(user_va_to_pa(page_dir, (void *) va));
    *pte_containing_va &= ~PTE_V;
  }
}
```

lab2_3 缺页异常

lab2_3 缺页异常需要在 PKE 操作系统内核中完善用户态栈空间的管理，使得它能够正确处理用户进程的“压栈”请求。具体思路如下：

1) 实验分析：在本实验中，我们需要处理应用程序执行过程中的缺页异常。与之前处理非法指令异常的情况不同，我们不应该将应用进程终止，而是应该动态分配物理页并将其映射到缺失的虚拟地址上。
2) 具体实现：如果缺页的逻辑地址在用户栈空间内，可以认为是合法的。可以给用户栈空间一个上限（本实验默认地址合法）。通过输入参数 stval 判断缺页的逻辑地址。根据合法性判断和需求，分配一个物理页。将分配的物理页映射到缺失的虚拟地址上，完成缺页处理。

具体实现的代码如下：

```
//
// the page fault handler. added @lab2_3. parameters:
// sepc: the pc when fault happens;
// stval: the virtual address that causes pagefault when being accessed.
//
void handle_user_page_fault(uint64 mcause, uint64 sepc, uint64 stval) {
```

```
7.    sprint("handle_page_fault: %lx\n", stval);
8.    switch (mcause) {
9.      case CAUSE_STORE_PAGE_FAULT:
10.       // TODO (lab2_3): implement the operations that solve the page
     fault to
11.       // dynamically increase application stack.
12.       // hint: first allocate a new physical page, and then, maps th
    e new page to the
13.       // virtual address that causes the page fault.
14.       // panic( "You need to implement the operations that actually
    handle the page fault in lab2_3.\n" );
15.
16.       user_vm_map((pagetable_t) current->pagetable, stval - stval %
    PGSIZE, PGSIZE, (uint64) alloc_page(), prot_to_type(PROT_WRITE | PRO
    T_READ, 1));
17.
18.       break;
19.     default:
20.       sprint("unknown page fault.\n");
21.       break;
22.   }
23.}
```

1.3 实验调试及心得

通过完成这三个内存管理相关的实验，我深入理解了操作系统内核中关键的内存管理概念和实现原理。在实验 1 中，我学到了如何实现虚实地址的转换，更加深入理解了页表项，页目录等核心概念。在实验 2 中，我深入研究了内存分配和回收的机制。在实验 3 中，通过处理缺页异常，我更深刻地理解了分页机制、页表操作和内存分配的重要性。

这些实验提供了丰富的调试和错误定位经验，同时也加强了我对内核模式切换、异常处理和内存管理的逻辑清晰性的要求。通过动手实践，我不仅深化了对操作系统内存管理的理论理解，还培养了解决实际问题的实践能力。

实验三 进程管理

1.1 实验目的

本实验旨在加深对进程管理的理解，并完善操作系统内核中的相关功能。具体目标包括：

1) 完善进程创建。
2) 实现主动释放 CPU。
3) 实现简单的轮转调度算法。

通过完成这些任务，将能够更深入地理解进程的创建、调度和管理，并对操作系统内核的设计和实现有更为全面的认识。

1.2 实验内容

lab3_1 进程创建（fork）

lab3_1 进程创建（fork）需要完善操作系统内核 kernel/process.c 文件中的 do_fork()函数。具体实现思路如下：

对于代码段，我们不直接复制，而是通过映射将子进程中的逻辑地址空间映射到父进程中装载代码段的物理页面。可以借助实验 2 中的 vm_map()函数来完成这一映射。

具体实现的代码如下：

```
case CODE_SEGMENT:
        // TODO (lab3_1): implment the mapping of child code segment to parent's
        // code segment.
        // hint: the virtual address mapping of code segment is tracked in mapped_info
        // page of parent's process structure. use the information in mapped_info to
        // retrieve the virtual to physical mapping of code segment.
        // after having the mapping information, just map the corresponding virtual
        // address region of child to the physical pages that actually store the code
        // segment of parent process.
        // DO NOT COPY THE PHYSICAL PAGES, JUST MAP THEM.
```

```
        // panic( "You need to implement the code segment mapping of child in lab3_1.\n" );

        for (int j = 0; j < parent->mapped_info[i].npages; j++) {
          uint64 pa_of_mapped_va = lookup_pa(parent->pagetable, parent->mapped_info[i].va + j * PGSIZE);
          map_pages(child->pagetable, parent->mapped_info[i].va + j * PGSIZE, PGSIZE, pa_of_mapped_va, prot_to_type(PROT_READ | PROT_EXEC, 1));
        }

        // after mapping, register the vm region (do not delete codes below!)
        child->mapped_info[child->total_mapped_region].va = parent->mapped_info[i].va;
        child->mapped_info[child->total_mapped_region].npages =
                parent->mapped_info[i].npages;
        child->mapped_info[child->total_mapped_region].seg_type = CODE_SEGMENT;
        child->total_mapped_region++;
        break;
```

lab3_2 进程 yield

lab3_2 进程 yield 需要完善 yield 系统调用，实现进程执行过程中的主动释放 CPU 的动作。具体思路如下：

1) 寻找 yield 系统调用，完善进程释放。
2) 进程释放：将当前进程置为就绪状态（READY），将当前进程加入到就绪队列的队尾，转进程调度。

具体实现的代码如下：

```
//
// kerenl entry point of yield. added @lab3_2
//
ssize_t sys_user_yield() {
  // TODO (lab3_2): implment the syscall of yield.
  // hint: the functionality of yield is to give up the processor. therefore,
  // we should set the status of currently running process to READY, insert it in
  // the rear of ready queue, and finally, schedule a READY process to run.
  //panic( "You need to implement the yield syscall in lab3_2.\n" );

```

```
11.   current->status = READY;
12.   insert_to_ready_queue(current);
13.   schedule();
14.   return 0;
15.}
```

lab3_3 循环轮转调度

lab3_3 循环轮转调度需要实现 kernel/strap.c 文件中的 rrsched()函数，获得预期结果。具体思路如下：

1) 确定时间片的定义：在 kernel/sched.h 文件中定义了“时间片”的长度，即为每隔两个 ticks 触发一次进程重新调度动作。
2) 实现循环调度逻辑：判断当前进程的 tick_count 加 1 后是否大于等于 TIME_SLICE_LEN，若答案为 yes，则应将当前进程的 tick_count 清零，并将当前进程加入就绪队列，转进程调度；若答案为 no，则应将当前进程的 tick_count 加 1，并返回。

具体实现的代码如下：

```
1. //
2. // implements round-robin scheduling. added @lab3_3
3. //
4. void rrsched() {
5.   // TODO (lab3_3): implements round-robin scheduling.
6.   // hint: increase the tick_count member of current process by one,
    if it is bigger than
7.   // TIME_SLICE_LEN (means it has consumed its time slice), change i
   ts status into READY,
8.   // place it in the rear of ready queue, and finally schedule next
   process to run.
9.   //panic( "You need to further implement the timer handling in lab3
   _3.\n" );
10.  if (++current->tick_count >= TIME_SLICE_LEN) {
11.    current->tick_count = 0;
12.    current->status = READY;
13.    insert_to_ready_queue(current);
14.    schedule();
15.  }
16.}
```

1.3 实验调试及心得

通过完成这三个进程相关的实验，我深刻理解了操作系统内核中进程管理的

重要性以及实现细节。在实验一中，我跟踪了从用户空间应用程序到内核的系统调用路径，深入了解了进程的创建过程；在实验二中，通过实现 yield 系统调用，我掌握了进程主动释放 CPU 的机制；同时，对简单的轮转调度算法的实现，加深了我对进程调度原理的理解。

整体而言，这些实验让我从不同角度深入了解了进程管理的方方面面，培养了我理解和处理复杂系统的能力。我不仅掌握了操作系统的核心概念，还提高了编码和调试的实际技能。

挑战实验 进程等待和数据段复制

1.1 实验目的

本实验通过修改 PKE 内核和系统调用，为用户程序提供 wait 函数的功能，wait 函数接受一个参数 pid：

当 pid 为-1 时，父进程等待任意一个子进程退出即返回子进程的 pid；

当 pid 大于 0 时，父进程等待进程号为 pid 的子进程退出即返回子进程的 pid；

如果 pid 不合法或 pid 大于 0 且 pid 对应的进程不是当前进程的子进程，返回-1。

补充 do_fork 函数，实验 3_1 实现了代码段的复制，你需要继续实现数据段的复制并保证 fork 后父子进程的数据段相互独立。

注意：最终测试程序可能和给出的用户程序不同，但都只涉及 wait 函数、fork 函数和全局变量读写的相关操作。

1.2 实验内容

本实验主要包含添加系统调用、在内核中实现 wait 函数的功能以及对 do_fork 函数的完善。

系统调用完善：需要补充完成对 wait()函数的系统调用流程，然后在 syscall.c 中完成系统调用函数的实现，wait()函数接受一个参数 pid:

1) 当 pid 为-1 时，父进程等待任意一个子进程退出即返回子进程的 pid；
2) 当 pid 大于 0 时，父进程等待进程号为 pid 的子进程退出即返回子进程的 pid；
3) 当 pid 不合法或者 pid 大于 0 且 pid 对应的进程不是当前进程的子进程，返回-1。

具体的代码实现如下：

```
1. //
2. // kerenl entry point of SYS_user_wait
3. //
4. ssize_t sys_user_wait(int pid) {
5.   int flag = 0;
6.   process *p = ready_queue_head;
7.
```

```
  //illegal format
  if(pid == 0 || pid < -1 || pid >= NPROC) flag = 0;
  if(flag) return -1;

  if (pid == -1) flag = 1;
  while(1){
    if (p->pid != pid){
      if(p -> queue_next == NULL) break;
      p = p->queue_next;
    }
    else{
      flag = 1;
      break;
    }
  };
  if (flag) {
    current->status = BLOCKED;
    schedule();
  }
  else return -1;

  return 0;
}
```

进一步完成 **do_fork()函数补充**，在 lab3_1 中只实现了代码段的复制，需要继续实现数据段的复制并保证 fork 后数据段相互独立。因此，数据段的内容就需要单独申请空间拷贝而不是像代码段一样进行简单的复制操作。

具体补全的代码如下：

```
// ! add for lab3_challenge1
      case DATA_SEGMENT:
        for (int j = 0; j < parent->mapped_info[i].npages; j++) {
          uint64 pa_of_mapped_va = lookup_pa(parent->pagetable, parent->mapped_info[i].va + j * PGSIZE);
          // need to register new page
          void *new_addr = alloc_page();
          memcpy(new_addr, (void *) pa_of_mapped_va, PGSIZE);
          map_pages(child->pagetable, parent->mapped_info[i].va + j * PGSIZE, PGSIZE, (uint64) new_addr, prot_to_type(PROT_READ | PROT_WRITE, 1));
        }

      // after mapping, register the vm region (do not delete codes below!)
```

```
        child->mapped_info[child->total_mapped_region].va = parent->mapped_info[i].va;
        child->mapped_info[child->total_mapped_region].npages =
                parent->mapped_info[i].npages;
        child->mapped_info[child->total_mapped_region].seg_type = DATA_SEGMENT;
        child->total_mapped_region++;
        break;
```

最后进行**进程状态转换的检查**，当进程的状态被设置为 BLOCKED 后，该进程就进入等待状态。然后当某一个进程执行完毕退出的时候，我们在该程序退出前进行检查，检查该程序是否有对应的父程序在阻塞，如果有，将该进程转为就绪态。在 exit()函数中的 free_process 完成，具体实现如下：

```
//
// reclaim a process. added @lab3_1
//
int free_process(process *proc) {
  // we set the status to ZOMBIE, but cannot destruct its vm space immediately.
  // since proc can be current process, and its user kernel stack is currently in use!
  // but for proxy kernel, it (memory leaking) may NOT be a really serious issue,
  // as it is different from regular OS, which needs to run 7x24.
  proc->status = ZOMBIE;

  // add for lab3_challenge1
  if (proc->parent != NULL && proc->parent->status == BLOCKED) {
    proc->parent->status = READY;
    insert_to_ready_queue(proc->parent);
  }

  return 0;
}
```

最终完成了挑战实验，得到了预期的输出结果。

1.3 实验调试及心得

通过完成 lab3_challenge1 的实验，我深刻理解了进程的创建、复制和等待的过程，以及对应的系统调用的实现。

首先，在补充完善 do_fork()函数时，我学到了如何合理处理父子进程的数据段，确保它们相互独立而不互相干扰。通过对每个进程的数据段进行申请空间拷贝，我确保了进程在运行时对数据的修改不会影响到其他进程其次，实现 wait()系统调用时，我深入理解了等待子进程退出的机制。通过判断 pid 的值，我实现了等待任意子进程退出或等待特定子进程退出的逻辑。最后，我学到了如何检查父进程是否阻塞在等待子进程退出的状态，并在子进程退出时将其转换为就绪态。

总的来说，通过这个实验，我不仅加深了对进程管理的理解，还提高了对系统调用的实际应用能力。