统一系统调用接口

郭振豪

2023/3/6——v1.0

本文档是在 mini-os v1.3.17 基础上构建的，对系统调用的架构进行了完善，使得系统调用的架构更加清晰，可以方便地根据提供的接口添加新的系统调用；对 mini-os 的测试程序进行了简单的梳理，标注了运行哪些测试用例可以测试系统调用，方便回归测试；另外，在 v1.3.17 的基础上增加了 PCB 的数量，以保证其有充足的 PCB 可供分配。

2023/3/20——v2.0

本次更新增加了汇编函数中对于寄存器进行保护的内容以及 exec 相关函数的调整。我将此版本定为 v1.3.18，其支持的环境如下：

- Ubuntu 20.04
- gcc 9.4.0
- nasm 2.14.02
- qemu-system-i386 4.2.1
- make 4.2.1

一、系统调用流程回顾

在谈论本次修改的具体细节之前，我们先对过去 mini-os 中系统调用实现的方式进行一下梳理，以帮助我们更好地理解。我们以系统调用 `read` 为例。

在用户态下，`read` 的函数声明如下：

```
int read(int fd, void* buf, int count);
```

函数定义如下：

```
read:
  push 3      ;the number of parameters
  mov ebx, esp
  mov eax, _NR_read
  int INT_VECTOR_SYS_CALL
  add esp, 4
  ret
```

当调用`read()`时，`read`的三个参数（count、buf、fd）及`read()`执行后需要返回的下一条指令的地址会被依次压入用户栈（这些操作由编译器实现）。

而在`read`的定义中，我们可以看到，汇编指令向用户栈中压入了参数个数（3），将用户栈当前的栈顶指针（esp）赋值给了ebx，将系统调用号（_NR_read）赋值给了 eax，随后执行了与中断向量号（INT_VECTOR_SYS_CALL）对应的系统调用中断处理函数（sys_call）。

注：INT_VECTOR_SYS_CALL 和 sys_call 已经提前被绑定在了中断向量表（IDT）中。

现在我们来看`sys_call`的实现：

```
sys_call:
;get syscall number from eax
;syscall that's called gets its argument from pushed ebx
;so we can't modify eax and ebx in save_syscall
  call  save_syscall    ;save registers and some other things.
  sti
  push  ebx             ;push the argument the syscall need
  call  [sys_call_table + eax * 4]  ;将参数压入堆栈后再调用函数
  add   esp, 4          ;clear the argument in the stack
  cli
  mov   edx, [p_proc_current]
  mov   esi, [edx + ESP_SAVE_SYSCALL]
  mov   [esi + EAXREG - P_STACKBASE], eax ;the return value of C function is in EAX
  ret
```

注：`int`指令会将用户栈切换至内核栈，故此处的栈是内核栈。

`sys_call`我们抓住重点，`call  [sys_call_table + eax * 4]`这条语句根据前文 eax 中保存的系统调用号，找到对应的系统调用函数并执行（此处是`sys_read()`）。那么`sys_read()`的参数是什么呢？可以看到这条语句之前将ebx压栈，而ebx中存入的是用户栈的栈顶指针，这就是`sys_read()`的参数。

来看一下`sys_read()`的具体实现：

```
PUBLIC int sys_read(void *uesp)
{
    return do_read(get_arg(uesp, 1), get_arg(uesp, 2), get_arg(uesp, 3));
}

PUBLIC int do_read(int fd, char *buf, int count) {
    return kern_read(fd, buf, count);
}

PUBLIC int kern_read(int fd, char *buf, int count) {
    // Some operation.
}
```

这里浅提一句，为了增强系统的可维护性，我们将内核态下的系统调用函数进行分层，以sys_xx() → do_xx() → kern_xx() 的形式进行调用。sys_xx() 中获取系统调用参数，do_xx() 中对有需要的参数进行一定的处理，kern_xx() 中是真正的处理逻辑。

由`sys_read()`的实现可以看出，其通过`get_arg`来获取系统调用参数：

```
; used to get the specified argument of the syscall from user space stack
; @uesp: user space stack pointer
; @order: which argument you want to get
; @uesp+0: the number of args, @uesp+8: the first arg, @uesp+12: the second arg...
get_arg:
  push ebp
  mov ebp, esp
  push esi
  push edi
  mov esi, dword [ebp + 8]      ;void *uesp
  mov edi, dword [ebp + 12]     ;int order
  mov eax, dword [esi + edi * 4 + 4]
  pop edi
```

```
  pop esi
  pop ebp
  ret
```

`get_arg`的实现很简单，根据偏移量从用户栈中取出参数。

在取出参数，执行完真正的系统调用处理逻辑后，返回`sys_call`，恢复现场，最后返回用户态。这样一次系统调用就执行完成了。

二、提出问题

回顾完 mini-os 系统调用的总体流程后，我们来讨论一下系统现在的问题。

现在 mini-os 的实现中，参数传递的方式并不统一。多参数的系统调用通过用户栈传递参数；单参数的系统调用部分通过用户栈传递参数，部分通过寄存器传递参数。上文我们以`read`系统调用为例，讨论了通过用户栈传递参数的方式，现在我们以`pthread_mutex_lock`为例，看看单参数的系统调用如何通过寄存器传递参数。

在用户态下，`pthread_mutex_lock`的函数声明如下：

```
int pthread_mutex_lock (pthread_mutex_t* mutex);
```

函数定义如下：

```
pthread_mutex_lock:
  mov ebx,[esp+4]
  mov eax, _NR_pthread_mutex_lock
  int INT_VECTOR_SYS_CALL
  ret
```

当调用`pthread_mutex_lock()`时，`pthread_mutex_lock`的一个参数（mutex）及`pthread_mutex_lock()`执行后需要返回的下一条指令的地址会被依次压入用户栈（这些操作由编译器实现）。

此时，esp 指向的是返回地址。而在`pthread_mutex_lock`的定义中，我们可以看到，ebx 被赋值为[esp+4]，也就是参数 mutex。eax被赋值为系统调用号（_NR_pthread_mutex_lock），随后执行了与中断向量号（INT_VECTOR_SYS_CALL）对应的系统调用中断处理函数（sys_call）。

我们重新回到sys_call的实现：

```
sys_call:
;get syscall number from eax
;syscall that's called gets its argument from pushed ebx
;so we can't modify eax and ebx in save_syscall
  call  save_syscall    ;save registers and some other things.
  sti
  push  ebx             ;push the argument the syscall need
  call  [sys_call_table + eax * 4]  ;将参数压入堆栈后再调用函数
  add   esp, 4          ;clear the argument in the stack
  cli
  mov   edx, [p_proc_current]
  mov   esi, [edx + ESP_SAVE_SYSCALL]
  mov   [esi + EAXREG - P_STACKBASE], eax ;the return value of C function is in EAX
  ret
```

此处依然将 ebx 压栈作为对应的系统调用函数（此处是`sys_pthread_mutex_lock()`）的参数，但是此次的 ebx 中存放着的不是用户栈的地址，而是`pthread_mutex_lock`的参数 mutex。

来看一下`sys_pthread_mutex_lock()`的具体实现：

```
int sys_pthread_mutex_lock(pthread_mutex_t *mutex) //阻塞式获取互斥变量
{
```

```
  // Some operation.
}
```

由于 mutex 直接作为参数，不需要从用户栈中获取，因此`sys_pthread_mutex_lock()`并没有调用`get_arg()`方法，直接进入处理逻辑，之后返回`sys_call`，恢复现场，最后返回用户态。

看到这里，想必大家已经对于系统调用接口不统一带来的繁复感有所体会，那么接下来我就着手简化这一流程。

三、修改

3.1 系统调用修改

3.1.1 用户态

如上文所见，过去 mini-os 的系统调用接口由汇编语言编写，定义在`syscall.asm`文件中（在本次修改中被弃用）。不同的系统调用采用的参数传递方式可能不同。因此，我们可以抽象出几个通用接口，统一通过寄存器传递参数。

```
/* 无参数的系统调用 */
#define _syscall0(NR_syscall) ({    \
  int retval;                \
  asm volatile (            \
    "int $0x90"           \
    : "=a" (retval)         \
    : "a" (NR_syscall)        \
    : "cc", "memory"        \
  );                    \
  retval;                   \
})

/* 一个参数的系统调用 */
#define _syscall1(NR_syscall, ARG1) ({    \
  int retval;                     \
  asm volatile (                  \
    "int $0x90"               \
    : "=a" (retval)             \
    : "a" (NR_syscall), "b" (ARG1)    \
    : "cc", "memory"          \
  );                        \
  retval;                     \
})

/* 两个参数的系统调用 */
#define _syscall2(NR_syscall, ARG1, ARG2) ({  \
  int retval;                      \
  asm volatile (                     \
  "int $0x90"                     \
    : "=a" (retval)                 \
    : "a" (NR_syscall), "b" (ARG1), "c" (ARG2)  \
    : "cc", "memory"            \
  );                          \
  retval;                       \
})

/* 三个参数的系统调用 */
#define _syscall3(NR_syscall, ARG1, ARG2, ARG3) ({  \
  int retval;                        \
  asm volatile (                       \
  "int $0x90"                       \
    : "=a" (retval)                   \
    : "a" (NR_syscall), "b" (ARG1), "c" (ARG2), "d" (ARG3)  \
    : "cc", "memory"                  \
  );                              \
  retval;                           \
})

/* 四个参数的系统调用 */
#define _syscall4(NR_syscall, ARG1, ARG2, ARG3, ARG4) ({  \
  int retval;                             \
  asm volatile (                            \
```

```
    "int $0x90"                         \
    : "=a" (retval)                       \
    : "a" (NR_syscall), "b" (ARG1), "c" (ARG2), "d" (ARG3), "S" (ARG4)  \
    : "cc", "memory"                      \
  );                                  \
  retval;                               \
})

/* 五个参数的系统调用 */
#define _syscall5(NR_syscall, ARG1, ARG2, ARG3, ARG4, ARG5) ({  \
  int retval;                                 \
  asm volatile (                                \
    "int $0x90"                           \
    : "=a" (retval)                         \
    : "a" (NR_syscall), "b" (ARG1), "c" (ARG2), "d" (ARG3), "S" (ARG4), "D" (ARG5)  \
    : "cc", "memory"                        \
  );                                    \
  retval;                                 \
})
```

这些接口根据参数的个数进行划分，将参数存放进对应的寄存器中（依次是 ebx，ecx， edx， esi， edi），依然将系统调用号（NR_syscall）存放进 eax 寄存器中，然后调用`int $0x90`，此处的 0x90 是系统调用对应的中断向量号，有些 OS 中设置的是 0x80。有了这些准备，我们就可以根据系统调用不同的参数个数，调用不同的接口。

▼ 系统调用定义

```
int get_ticks() {
  return _syscall0(_NR_get_ticks);
}

int get_pid() {
  return _syscall0(_NR_get_pid);
}
void* malloc_4k() {
  return _syscall0(_NR_malloc_4k);
}

int free_4k(void* AdddrLin) {
  return _syscall1(_NR_free_4k, AdddrLin);
}

int fork() {
  return _syscall0(_NR_fork);
}

int pthread_create(int* thread, void* attr, void* entry, void* arg) {
  return _syscall4(_NR_pthread_create, thread, attr, entry, arg);
}

void udisp_int(int arg) {
  _syscall1(_NR_udisp_int, arg);
}

void udisp_str(char* arg) {
  _syscall1(_NR_udisp_str, arg);
}

u32 exec(char* path, char* argv[], char* envp[]) {
  return _syscall3(_NR_exec, path, argv, envp);
}

void yield() {
  _syscall0(_NR_yield);
}

void sleep(int n) {
  return _syscall1(_NR_sleep, n);
}

int open(const char* pathname, int flags) {
  return _syscall2(_NR_open, pathname, flags);
}
```

```
int close(int fd) {
  return _syscall1(_NR_close, fd);
}

int read(int fd, void* buf, int count) {
  return _syscall3(_NR_read, fd, buf, count);
}

int write(int fd, const void* buf, int count) {
  return _syscall3(_NR_write, fd, buf, count);
}

int lseek(int fd, int offset, int whence) {
  return _syscall3(_NR_lseek, fd, offset, whence);
}

int unlink(const char* pathname) {
  return _syscall1(_NR_unlink, pathname);
}

int create(char* pathname) {
  return _syscall1(_NR_create, pathname);
}

int delete(const char* pathname) {
  return _syscall1(_NR_delete, pathname);
}

int opendir(const char* dirname) {
  return _syscall1(_NR_opendir, dirname);
}

int createdir(const char* dirname) {
  return _syscall1(_NR_createdir, dirname);
}

int deletedir(const char* dirname) {
  return _syscall1(_NR_deletedir, dirname);
}

int readdir(const char* dirname, unsigned int dir[3], char* filename) {
  return _syscall3(_NR_readdir, dirname, dir, filename);
}

int chdir(const char* path) {
  return _syscall1(_NR_chdir, path);
}

char* getcwd(char* buf, int size) {
  return _syscall2(_NR_getcwd, buf, size);
}

int wait_() {
  return _syscall0(_NR_wait);
}

void exit(int status) {
  _syscall1(_NR_exit, status);
}

// "user/ulib/signal.c" 中提供了上层封装
int _signal(int sig, void* handler, void* _Handler) {
  return _syscall3(_NR_signal, sig, handler, _Handler);
}

int sigsend(int pid, Sigaction* sigaction_p) {
  return _syscall2(_NR_sigsend, pid, sigaction_p);
}

void sigreturn(int ebp) {
  _syscall1(_NR_sigreturn, ebp);
}

u32 total_mem_size() {
  return _syscall0(_NR_total_mem_size);
}
```

```
int shmget(int key, int size, int shmflg) {
  return _syscall3(_NR_shmget, key, size, shmflg);
}

// "user/ulib/ushm.c" 中提供了上层封装
void* _shmat(int shmid, char* shmaddr, int shmflg) {
  return _syscall3(_NR_shmat, shmid, shmaddr, shmflg);
}

// "user/ulib/ushm.c" 中提供了上层封装
void _shmdt(char* shmaddr) {
  _syscall1(_NR_shmdt, shmaddr);
}

struct ipc_shm* shmctl(int shmid, int cmd, struct ipc_shm* buf) {
  return _syscall3(_NR_shmctl, shmid, cmd, buf);
}

void* shmmemcpy(void* dst, const void* src, long unsigned int len) {
  return _syscall3(_NR_shmmemcpy, dst, src, len);
}

int ftok(char* f,int key) {
  return _syscall2(_NR_ftok, f, key);
}

int msgget(key_t key, int msgflg) {
  return _syscall2(_NR_msgget, key, msgflg);
}

int msgsnd(int msqid, const void* msgp, int msgsz, int msgflg) {
  return _syscall4(_NR_msgsnd, msqid, msgp, msgsz, msgflg);
}

int msgrcv(int msqid, void* msgp, int msgsz, long msgtyp, int msgflg) {
  return _syscall5(_NR_msgrcv, msqid, msgp, msgsz, msgtyp, msgflg);
}

int msgctl(int msgqid, int cmd, msqid_ds* buf) {
  return _syscall3(_NR_msgctl, msgqid, cmd, buf);
}

void test(int no) {
  _syscall1(_NR_test, no);
}

u32 execvp(char* file, char* argv[]) {
  return _syscall2(_NR_execvp, file, argv);
}

u32 execv(char* path, char* argv[]) {
  return _syscall2(_NR_execv, path, argv);
}

pthread_t pthread_self() {
  return _syscall0(_NR_pthread_self);
}

int pthread_mutex_init (pthread_mutex_t* mutex, pthread_mutexattr_t* mutexattr) {
  return _syscall2(_NR_pthread_mutex_init, mutex, mutexattr);
}

int pthread_mutex_destroy(pthread_mutex_t* mutex) {
  return _syscall1(_NR_pthread_mutex_destroy, mutex);
}

int pthread_mutex_lock (pthread_mutex_t* mutex) {
  return _syscall1(_NR_pthread_mutex_lock, mutex);
}

int pthread_mutex_unlock (pthread_mutex_t* mutex) {
  return _syscall1(_NR_pthread_mutex_unlock, mutex);
}

int pthread_mutex_trylock(pthread_mutex_t* mutex) {
```

```
  return _syscall1(_NR_pthread_mutex_trylock, mutex);
}

int pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* cond_attr) {
  return _syscall2(_NR_pthread_cond_init, cond, cond_attr);
}

int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex) {
  return _syscall2(_NR_pthread_cond_wait, cond, mutex);
}

int pthread_cond_timewait(pthread_cond_t* cond, pthread_mutex_t* mutex, int* timeout) {
  return _syscall3(_NR_pthread_cond_timewait, cond, mutex, timeout);
}

int pthread_cond_signal(pthread_cond_t* cond) {
  return _syscall1(_NR_pthread_cond_signal, cond);
}

int pthread_cond_broadcast(pthread_cond_t* cond) {
  return _syscall1(_NR_pthread_cond_broadcast, cond);
}

int pthread_cond_destroy(pthread_cond_t* cond) {
  return _syscall1(_NR_pthread_cond_destroy, cond);
}

int get_pid_byname(char* name) {
  return _syscall1(_NR_get_pid_byname, name);
}

int mount(const char *source, const char *target,const char *filesystemtype,
      unsigned long mountflags, const void *data) {
  return _syscall5(_NR_mount, source, target, filesystemtype, mountflags, data);
}

int umount(const char *target) {
  return _syscall1(_NR_umount, target);
}

int init_block_dev(int drive) {
  return _syscall1(_NR_init_block_dev, drive);
}
```

3.1.2 内核态

当从用户态陷入内核态后，用户栈也切换到了内核栈，这时如何获取参数成了我们需要考虑的问题。在用户态下，我们通过寄存器传递参数，因此在内核态下，我们完全可以从寄存器中取出参数。但由于在 OS 的持续运行当中，寄存器的值可能会发生改变，因此我们可以将寄存器的值压入内核栈中，进行保存。刚好，`sys_call`的第一条指令调用的`save_syscall`函数中就对上下文环境进行了保存。

```
save_syscall:
  pushad          ; \
  push    ds      ;  |
  push    es      ;  | 保存原寄存器值
  push    fs      ;  |
  push    gs      ; /
  ; Some Operation
```

在将寄存器保存在内核栈之后，我们如何从内核栈中获取这些参数呢？最直观的一个想法就是通过一个指针指向当前内核栈栈顶的位置，通过偏移量获取对应的参数。那么一个新问题又来了：这个指针应该存放在哪里呢？如果设置为全局变量，那么随着进程的切换，该变量很有可能被修改。因此，该指针应该是每个进程私有的，存放在 PCB 中最适合不过。

```
typedef struct s_proc {
  STACK_FRAME regs;           // process registers saved in stack frame
  u16 ldt_sel;                // gdt selector giving ldt base and limit
  DESCRIPTOR ldts[LDT_SIZE];  // local descriptors for code and data
  char* esp_save_int;         // to save the position of esp in the kernel stack of the process
```

```
  char* esp_save_syscall;      // to save the position of esp in the kernel stack of the process
  char* esp_save_context;      // to save the position of esp in the kernel stack of the process

  char* esp_save_syscall_arg;  // NEW

  // Other variables
}PROCESS_0;
```

在 PCB 中，我们设置了指针变量`esp_save_syscall_arg`，接着在`save_syscall`中让该指针指向当前内核栈栈顶的位置。

```
save_syscall:
  pushad          ; `.
  push    ds      ;  |
  push    es      ;  | 保存原寄存器值
  push    fs      ;  |
  push    gs      ; /
  mov     edx,  [p_proc_current]
  mov     dword [edx + ESP_SAVE_SYSCALL_ARG], esp
```

注：偏移量`ESP_SAVE_SYSCALL_ARG`定义在`sconst.inc`文件中。

真正获取参数的函数`get_arg`需要修改为通过内核栈获取参数：

```
PUBLIC u32 get_arg(int order)
{
  STACK_FRAME* syscall_esp = (STACK_FRAME*)(p_proc_current->task.esp_save_syscall_arg);
  switch (order) {
    case 1:
      return syscall_esp->ebx;
    case 2:
      return syscall_esp->ecx;
    case 3:
      return syscall_esp->edx;
    case 4:
      return syscall_esp->esi;
    case 5:
      return syscall_esp->edi;
    default:
      disp_str("invalid order!");
  }
}

typedef struct s_stackframe { /* proc_ptr points here       ↑ Low     */
  u32 gs;     /* ┓            |     */
  u32 fs;     /* ┃            |     */
  u32 es;     /* ┃            |     */
  u32 ds;     /* ┃            |     */
  u32 edi;    /* ┃            |     */
  u32 esi;    /* ┣ pushed by save()      |     */
  u32 ebp;    /* ┃            |     */
  u32 kernel_esp; /* <- 'popad' will ignore it      |     */
  u32 ebx;    /* ┃            ↑栈从高地址往低地址增长*/
  u32 edx;    /* ┃            |     */
  u32 ecx;    /* ┃            |     */
  u32 eax;    /* ┛            |     */
  u32 retaddr;  /* return address for assembly code save()  |     */
  u32 eip;    /*  ┓           |     */
  u32 cs;     /*  ┃           |     */
  u32 eflags;   /*  ┣ these are pushed by CPU during interrupt  |     */
  u32 esp;    /*  ┃           |     */
  u32 ss;     /*  ┛           ┗High     */
}STACK_FRAME;
```

在通过当前 PCB 获取到`esp_save_syscall_arg`指针后，可以将其强制类型转换成`STACK_FRAME*`类型，进而方便地取出参数。

到这里，主体部分就修改完成了，但还是有一些冗余的地方。再次回顾一下`sys_call`：

```
sys_call:
;get syscall number from eax
;syscall that's called gets its argument from pushed ebx
;so we can't modify eax and ebx in save_syscall
  call  save_syscall    ;save registers and some other things.
  sti
  push  ebx             ;push the argument the syscall need
  call  [sys_call_table + eax * 4]  ;将参数压入堆栈后再调用函数
  add   esp, 4          ;clear the argument in the stack
  cli
  mov   edx, [p_proc_current]
  mov   esi, [edx + ESP_SAVE_SYSCALL]
  mov   [esi + EAXREG - P_STACKBASE], eax ;the return value of C function is in EAX
  ret
```

在修改之前，ebx 的作用有两个：

1. 当采用通过用户栈传递参数的方式时，ebx 中存放着用户栈栈顶的地址，作为参数传递给系统调用对应的处理函数。
2. 当采用通过寄存器传递参数的方式时，ebx 中存放着单个参数，同样作为参数传递给系统调用对应的处理函数。

在统一了系统调用的传参方式后，ebx 的这两个作用都不复存在，因此上文两条语句可以删去。

既然参数都通过内核栈进行传递，那么sys_xx()不再需要参数。仍旧以`sys_read()`为例，可以从左边更改为右边的形式。

```
PUBLIC int sys_read(void *uesp)
{
  return do_read(get_arg(uesp,1), get_arg(uesp,2), get_arg(uesp,3));
}

PUBLIC int do_read(int fd, char *buf, int count) {
  return kern_read(fd, buf, count);
}

PUBLIC int kern_read(int fd, char *buf, int count) {
  // Some operation.
}
```

```
PUBLIC int sys_read()
{
  return do_read(get_arg(1), get_arg(2), get_arg(3));
}

PUBLIC int do_read(int fd, char *buf, int count) {
  return kern_read(fd, buf, count);
}

PUBLIC int kern_read(int fd, char *buf, int count) {
  // Some operation.
}
```

至此，对系统调用整体的修改就完成了。

3.2 PCB 数量修改

`main.c`的`initialize_processes()`函数先后在下方四段代码块中对 PCB 进行了初始化。

```
for (pid = 0; pid < NR_TASKS; pid++) {}

for (; pid < NR_K_PCBS; pid++) {}

for (; pid < NR_K_PCBS + 1; pid++) {}

for (; pid < NR_PCBS; pid++) {}
```

据此可知，不同类型的 PCB 数量由`NR_TASKS`、`NR_K_PCBS`、`NR_PCBS`三个宏进行切割。因此在`proc.h`中进行修改：

```
#define NR_PCBS     64                                  //modified by zhenhao 2023.3.5
//#define NR_TASKS  4   //TestA~TestC + hd_service //deleted by mingxuan 2019-5-19
#define NR_TASKS    2   //task_tty + hd_service     //modified by mingxuan 2019-5-19
#define NR_K_PCBS   16                                  //modified by zhenhao 2023.3.5
```

修改后，PCB类型分布如下：

- pid 0~1，分配给 task_tty 和 hd_service
- pid 2~15，系统级进程，空闲
- pid 16，分配给 initial 进程
- pid 16~64，用户级进程，空闲

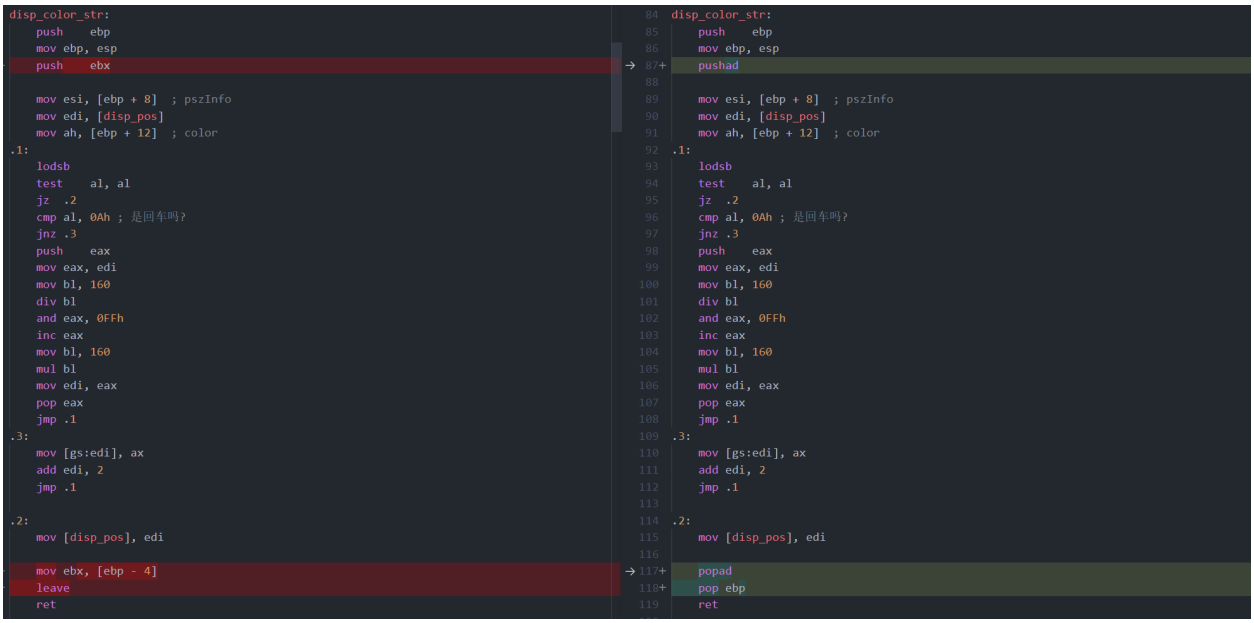
3.3 寄存器保护

`os/kernel/kernel.asm`

```
refresh_page_cache:                          798  refresh_page_cache:
                                             799+     push eax
    mov eax,cr3                              800      mov eax,cr3
    mov cr3,eax                              801      mov cr3,eax
                                             802+     pop eax
    ret                                      803      ret
```

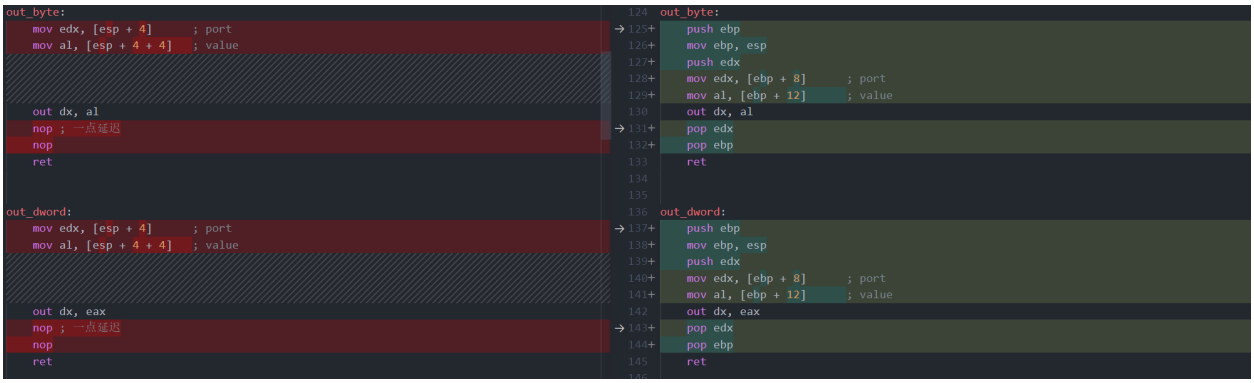
`os/klib/kliba.asm`

```
disp_str:                                    36  disp_str:
    push    ebp                              37      push    ebp
    mov ebp, esp                             38      mov ebp, esp
    push    ebx                              39+     pushad
                                             40
                                             41
    mov esi, [ebp + 8]  ; pszInfo            42      mov esi, [ebp + 8]  ; pszInfo
    mov edi, [disp_pos]                      43      mov edi, [disp_pos]
    mov ah, 0Fh                              44      mov ah, 0Fh
.1:                                          45  .1:
    lodsb                                    46      lodsb
    test    al, al                           47      test    al, al
    jz  .2                                   48      jz  .2
    cmp al, 0Ah ; 是回车吗?                  49      cmp al, 0Ah ; 是回车吗?
    jnz .3                                   50      jnz .3
    push    eax                              51      push    eax
    mov eax, edi                             52      mov eax, edi
    mov bl, 160                              53      mov bl, 160
    div bl                                   54      div bl
    and eax, 0FFh                            55      and eax, 0FFh
    inc eax                                  56      inc eax
    mov bl, 160                              57      mov bl, 160
    mul bl                                   58      mul bl
    mov edi, eax                             59      mov edi, eax
    pop eax                                  60      pop eax
    jmp .1                                   61      jmp .1
.3:                                          62  .3:
;added by xw, 17/12/11                       63  ;added by xw, 17/12/11
;added begin                                 64  ;added begin
    cmp edi, 1F40h                           65      cmp edi, 1F40h
    jnz .4                                   66      jnz .4
    mov edi, 0FA0h                           67      mov edi, 0FA0h
.4:                                          68  .4:
;added end                                   69  ;added end
    mov [gs:edi], ax                         70      mov [gs:edi], ax
    add edi, 2                               71      add edi, 2
    jmp .1                                   72      jmp .1
                                             73
.2:                                          74  .2:
    mov [disp_pos], edi                      75      mov [disp_pos], edi
                                             76
    mov ebx, [ebp - 4]                       77+     popad
    leave                                    78+     pop ebp
    ret                                      79      ret
```



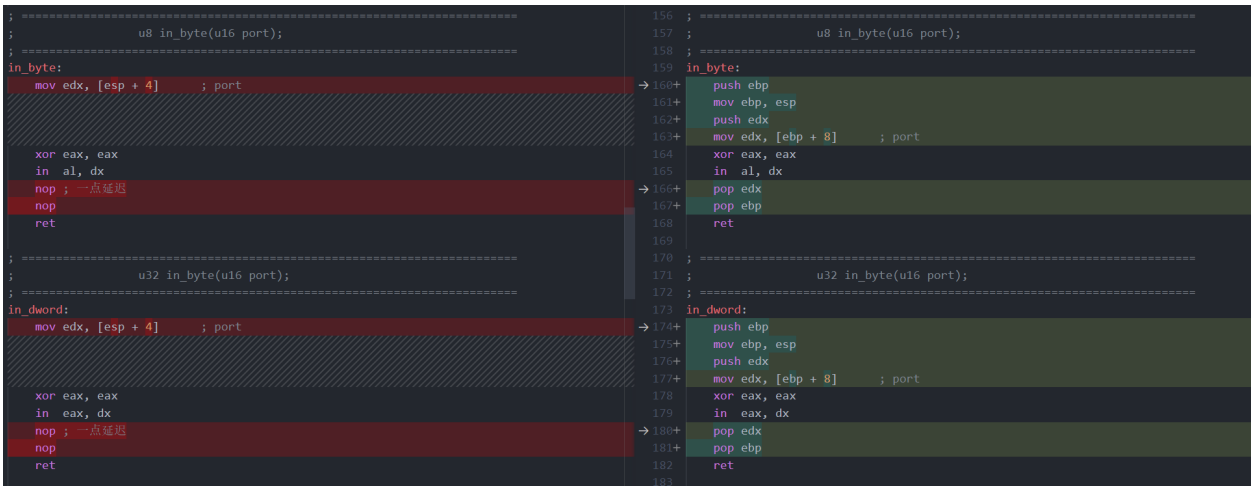
```
disp_color_str:
    push    ebp
    mov ebp, esp
    push    ebx

    mov esi, [ebp + 8]  ; pszInfo
    mov edi, [disp_pos]
    mov ah, [ebp + 12]  ; color
.1:
    lodsb
    test    al, al
    jz  .2
    cmp al, 0Ah ; 是回车吗?
    jnz .3
    push    eax
    mov eax, edi
    mov bl, 160
    div bl
    and eax, 0FFh
    inc eax
    mov bl, 160
    mul bl
    mov edi, eax
    pop eax
    jmp .1
.3:
    mov [gs:edi], ax
    add edi, 2
    jmp .1

.2:
    mov [disp_pos], edi

    mov ebx, [ebp - 4]
    leave
    ret
```

```
disp_color_str:
    push    ebp
    mov ebp, esp
    pushad

    mov esi, [ebp + 8]  ; pszInfo
    mov edi, [disp_pos]
    mov ah, [ebp + 12]  ; color
.1:
    lodsb
    test    al, al
    jz  .2
    cmp al, 0Ah ; 是回车吗?
    jnz .3
    push    eax
    mov eax, edi
    mov bl, 160
    div bl
    and eax, 0FFh
    inc eax
    mov bl, 160
    mul bl
    mov edi, eax
    pop eax
    jmp .1
.3:
    mov [gs:edi], ax
    add edi, 2
    jmp .1

.2:
    mov [disp_pos], edi

    popad
    pop ebp
    ret
```



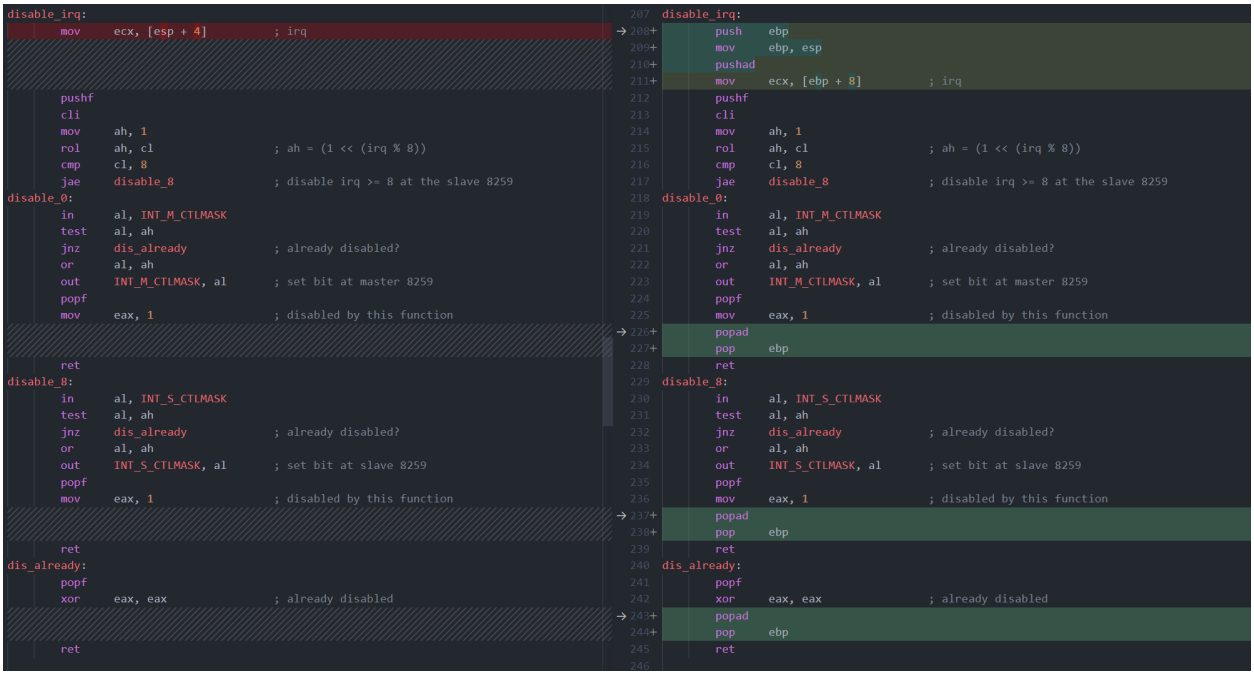
```
out_byte:
    mov edx, [esp + 4]      ; port
    mov al, [esp + 4 + 4]   ; value

    out dx, al
    nop ; 一点延迟
    nop
    ret


out_dword:
    mov edx, [esp + 4]      ; port
    mov al, [esp + 4 + 4]   ; value

    out dx, eax
    nop ; 一点延迟
    nop
    ret
```

```
out_byte:
    push ebp
    mov ebp, esp
    push edx
    mov edx, [ebp + 8]      ; port
    mov al, [ebp + 12]      ; value
    out dx, al
    pop edx
    pop ebp
    ret


out_dword:
    push ebp
    mov ebp, esp
    push edx
    mov edx, [ebp + 8]      ; port
    mov al, [ebp + 12]      ; value
    out dx, eax
    pop edx
    pop ebp
    ret
```



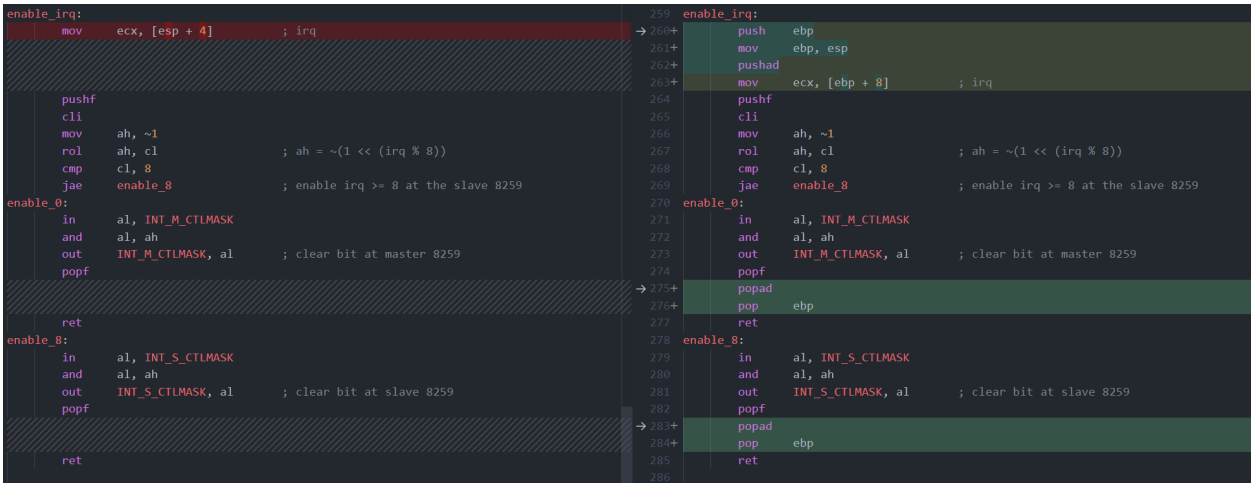
```
; ========================================================================
;                   u8 in_byte(u16 port);
; ========================================================================
in_byte:
    mov edx, [esp + 4]      ; port

    xor eax, eax
    in  al, dx
    nop ; 一点延迟
    nop
    ret

; ========================================================================
;                   u32 in_byte(u16 port);
; ========================================================================
in_dword:
    mov edx, [esp + 4]      ; port

    xor eax, eax
    in  eax, dx
    nop ; 一点延迟
    nop
    ret
```

```
; ========================================================================
;                   u8 in_byte(u16 port);
; ========================================================================
in_byte:
    push ebp
    mov ebp, esp
    push edx
    mov edx, [ebp + 8]      ; port
    xor eax, eax
    in  al, dx
    pop edx
    pop ebp
    ret

; ========================================================================
;                   u32 in_byte(u16 port);
; ========================================================================
in_dword:
    push ebp
    mov ebp, esp
    push edx
    mov edx, [ebp + 8]      ; port
    xor eax, eax
    in  eax, dx
    pop edx
    pop ebp
    ret
```



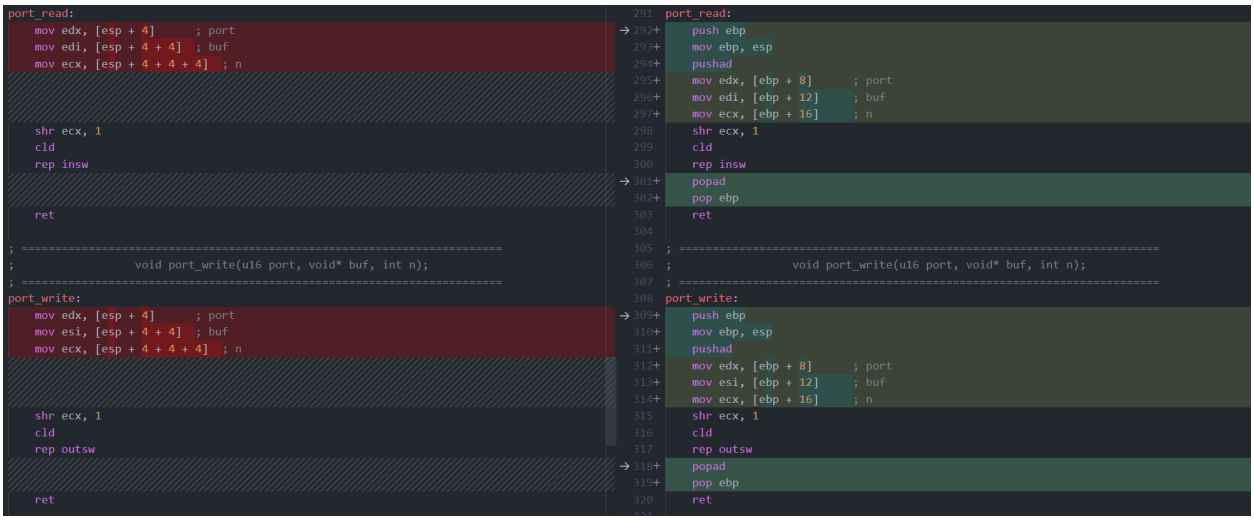
```
disable_irq:
        mov     ecx, [esp + 4]          ; irq
        pushf
        cli
        mov     ah, 1
        rol     ah, cl                  ; ah = (1 << (irq % 8))
        cmp     cl, 8
        jae     disable_8               ; disable irq >= 8 at the slave 8259
disable_0:
        in      al, INT_M_CTLMASK
        test    al, ah
        jnz     dis_already             ; already disabled?
        or      al, ah
        out     INT_M_CTLMASK, al       ; set bit at master 8259
        popf
        mov     eax, 1                  ; disabled by this function
        ret
disable_8:
        in      al, INT_S_CTLMASK
        test    al, ah
        jnz     dis_already             ; already disabled?
        or      al, ah
        out     INT_S_CTLMASK, al       ; set bit at slave 8259
        popf
        mov     eax, 1                  ; disabled by this function
        ret
dis_already:
        popf
        xor     eax, eax                ; already disabled
        ret
```

```
207 disable_irq:
208+        push    ebp
209+        mov     ebp, esp
210+        pushad
211+        mov     ecx, [ebp + 8]          ; irq
212         pushf
213         cli
214         mov     ah, 1
215         rol     ah, cl                  ; ah = (1 << (irq % 8))
216         cmp     cl, 8
217         jae     disable_8               ; disable irq >= 8 at the slave 8259
218 disable_0:
219         in      al, INT_M_CTLMASK
220         test    al, ah
221         jnz     dis_already             ; already disabled?
222         or      al, ah
223         out     INT_M_CTLMASK, al       ; set bit at master 8259
224         popf
225         mov     eax, 1                  ; disabled by this function
226+         popad
227+         pop     ebp
228         ret
229 disable_8:
230         in      al, INT_S_CTLMASK
231         test    al, ah
232         jnz     dis_already             ; already disabled?
233         or      al, ah
234         out     INT_S_CTLMASK, al       ; set bit at slave 8259
235         popf
236         mov     eax, 1                  ; disabled by this function
237+         popad
238+         pop     ebp
239         ret
240 dis_already:
241         popf
242         xor     eax, eax                ; already disabled
243+         popad
244+         pop     ebp
245         ret
246
```



```
enable_irq:
        mov     ecx, [esp + 4]          ; irq
        pushf
        cli
        mov     ah, ~1
        rol     ah, cl                  ; ah = ~(1 << (irq % 8))
        cmp     cl, 8
        jae     enable_8                ; enable irq >= 8 at the slave 8259
enable_0:
        in      al, INT_M_CTLMASK
        and     al, ah
        out     INT_M_CTLMASK, al       ; clear bit at master 8259
        popf
        ret
enable_8:
        in      al, INT_S_CTLMASK
        and     al, ah
        out     INT_S_CTLMASK, al       ; clear bit at slave 8259
        popf
        ret
```

```
259 enable_irq:
260+        push    ebp
261+        mov     ebp, esp
262+        pushad
263+        mov     ecx, [ebp + 8]          ; irq
264         pushf
265         cli
266         mov     ah, ~1
267         rol     ah, cl                  ; ah = ~(1 << (irq % 8))
268         cmp     cl, 8
269         jae     enable_8                ; enable irq >= 8 at the slave 8259
270 enable_0:
271         in      al, INT_M_CTLMASK
272         and     al, ah
273         out     INT_M_CTLMASK, al       ; clear bit at master 8259
274         popf
275+         popad
276+         pop     ebp
277         ret
278 enable_8:
279         in      al, INT_S_CTLMASK
280         and     al, ah
281         out     INT_S_CTLMASK, al       ; clear bit at slave 8259
282         popf
283+         popad
284+         pop     ebp
285         ret
286
```



```
port_read:
    mov edx, [esp + 4]      ; port
    mov edi, [esp + 4 + 4]  ; buf
    mov ecx, [esp + 4 + 4 + 4]  ; n
    shr ecx, 1
    cld
    rep insw
    ret

; ========================================================================
;                 void port_write(u16 port, void* buf, int n);
; ========================================================================
port_write:
    mov edx, [esp + 4]      ; port
    mov esi, [esp + 4 + 4]  ; buf
    mov ecx, [esp + 4 + 4 + 4]  ; n
    shr ecx, 1
    cld
    rep outsw
    ret
```

```
291 port_read:
292+    push ebp
293+    mov ebp, esp
294+    pushad
295+    mov edx, [ebp + 8]      ; port
296+    mov edi, [ebp + 12]     ; buf
297+    mov ecx, [ebp + 16]     ; n
298     shr ecx, 1
299     cld
300     rep insw
301+     popad
302+     pop ebp
303     ret
304
305 ; ========================================================================
306 ;                 void port_write(u16 port, void* buf, int n);
307 ; ========================================================================
308 port_write:
309+     push ebp
310+     mov ebp, esp
311+     pushad
312+     mov edx, [ebp + 8]      ; port
313+     mov esi, [ebp + 12]     ; buf
314+     mov ecx, [ebp + 16]     ; n
315     shr ecx, 1
316     cld
317     rep outsw
318+     popad
319+     pop ebp
320     ret
```

```asm
write_char:
    push    ebp
    mov ebp,esp

    mov esi,[ebp+8]
    mov edi,[disp_pos]

    push eax
    mov eax,esi
    mov ah, 0Fh
    mov [gs:edi], ax
    pop eax
    pop ebp
    ret
```

```asm
write_char:
    push    ebp
    mov ebp,esp
    pushad

    mov esi,[ebp+8]
    mov edi,[disp_pos]

    mov eax,esi
    mov ah, 0Fh
    mov [gs:edi], ax
    popad
    pop ebp
    ret
```

`os/klib/string.asm` 和 `user/ulib/string.asm`

```asm
strcpy:
    push    ebp
    mov     ebp, esp

    mov     esi, [ebp + 12] ; Source
    mov     edi, [ebp + 8]  ; Destination

.1:
    mov     al, [esi]               ; ┐
    inc     esi                     ; │
                    ; ├ 逐字节移动
    mov     byte [edi], al          ; │
    inc     edi                     ; ┘

    cmp     al, 0           ; 是否遇到 '\0'
    jnz     .1              ; 没遇到就继续循环，遇到就结束

    mov     eax, [ebp + 8]  ; 返回值

    pop     ebp
    ret                     ; 函数结束，返回
```

```asm
strcpy:
    push    ebp
    mov     ebp, esp
    pushad      You, 现在 • Uncommitted changes

    mov     esi, [ebp + 12] ; Source
    mov     edi, [ebp + 8]  ; Destination

.1:
    mov     al, [esi]               ; ┐
    inc     esi                     ; │
                    ; ├ 逐字节移动
    mov     byte [edi], al          ; │
    inc     edi                     ; ┘

    cmp     al, 0           ; 是否遇到 '\0'
    jnz     .1              ; 没遇到就继续循环，遇到就结束

    mov     eax, [ebp + 8]  ; 返回值

    popad
    pop     ebp
    ret                     ; 函数结束，返回
```

```asm
strlen:
        push    ebp
        mov     ebp, esp

        mov     eax, 0                  ; 字符串长度开始是 0
        mov     esi, [ebp + 8]          ; esi 指向首地址

.1:
        cmp     byte [esi], 0           ; 看 esi 指向的字符是否是 '\0'
        jz      .2                      ; 如果是 '\0'，程序结束
        inc     esi                     ; 如果不是 '\0'，esi 指向下一个字符
        inc     eax                     ;         并且，eax 自加一
        jmp     .1                      ; 如此循环

.2:
        pop     ebp
        ret                             ; 函数结束，返回
; ~zcr ------------------------------------------------------------
```

```asm
strlen:
        push    ebp
        mov     ebp, esp
        push    esi

        mov     eax, 0                  ; 字符串长度开始是 0
        mov     esi, [ebp + 8]          ; esi 指向首地址

.1:
        cmp     byte [esi], 0           ; 看 esi 指向的字符是否是 '\0'
        jz      .2                      ; 如果是 '\0'，程序结束
        inc     esi                     ; 如果不是 '\0'，esi 指向下一个字符
        inc     eax                     ;         并且，eax 自加一
        jmp     .1                      ; 如此循环

.2:
        pop     esi
        pop     ebp
        ret                             ; 函数结束，返回
; ~zcr ------------------------------------------------------------
```

3.4 exec 修改

由于Linux中并没有exec()系统调用，而是采用execve()，因此本次修改中将exec更名为execve。

四、测试

我对 mini-os 的测试程序进行了简单的梳理，标注了运行哪些测试用例可以测试系统调用，方便回归测试。后文被勾选部分的系统调用都经过我的测试，可用于测试的测试用例被标注在下方。未被勾选的系统调用暂时未在 mini-os 中找到可运行的测试用例。

- [ ] int get_ticks();
- [x] ~~int get_pid();~~

  运行`ptest1.bin`可调用，其源程序是`fortest1.c`。
- [ ] void* malloc_4k();
- [ ] int free_4k(void* AdddrLin);

- [x] ~~int fork();~~

  运行`test_4.bin`可调用，其源程序是`test_4.c`。
- [x] ~~int pthread_create(int* thread, void* attr, void* entry, void* arg);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~void udisp_int(int arg);~~

  运行`test_4.bin`可调用，其源程序是`test_4.c`。
- [x] ~~void udisp_str(char* arg);~~

  运行`test_4.bin`可调用，其源程序是`test_4.c`。
- [x] ~~u32 exec(char* path, char* argv[], char* envp[]);~~

  每成功执行一个 .bin 文件，都说明`exec`被成功调用了
- [x] ~~void yield();~~
  - 将ktest.c中现有的initial()注释掉，将"Syscall Yield Test"下的TestA()、TestB()、TestC()、initial()取消注释。
  - 将global.c中task_table里TestA、TestB、TestC的注释取消，将hd_service、task_tty部分注释掉。
  - 将proc.h中的NR_TASKS修改为3。
  - 编译后运行程序。
- [x] ~~void sleep(int n);~~

  运行`ptest4.bin`可调用，其源程序是`fortest4.c`。
- [x] ~~int open(const char* pathname, int flags);~~

  在`init.c`中有调用。
- [x] ~~int close(int fd);~~

  运行`test_1.bin`可调用，其源程序是`test_1.c`。
- [x] ~~int read(int fd, void* buf, int count);~~

  运行`test_1.bin`可调用，其源程序是`test_1.c`。
- [x] ~~int write(int fd, const void* buf, int count);~~

  运行`test_1.bin`可调用，其源程序是`test_1.c`。
- [ ] int lseek(int fd, int offset, int whence);
- [ ] int unlink(const char* pathname);
- [ ] int create(char* pathname);
- [ ] int delete(const char* pathname);
- [ ] int opendir(const char* dirname);
- [x] ~~int createdir(const char* dirname);~~

  注释掉`init.c`当前的`main`函数，取消"验证多个FAT32"下的`main`函数的注释，运行程序后输入`fat32_test`可调用。
- [x] ~~int deletedir(const char* dirname);~~

  注释掉`init.c`当前的`main`函数，取消"验证多个FAT32"下的`main`函数的注释，运行程序后输入`rmdir`可调用。
- [x] ~~int readdir(const char* dirname, unsigned int dir[3], char* filename);~~

  注释掉`init.c`当前的`main`函数，取消"验证多个FAT32"下的`main`函数的注释，运行程序后输入`ls`可调用。
- [x] ~~int chdir(const char* path);~~

注释掉`init.c`当前的`main`函数，取消“验证多个FAT32”下的`main`函数的注释，运行程序后输入`fat32_test`可调用。

- [x] ~~char* getcwd(char* buf, int size);~~

  注释掉`init.c`当前的`main`函数，取消“验证多个FAT32”下的`main`函数的注释，运行程序后输入`fat32_test`可调用。

- [x] ~~int wait_();~~

  在`shell_0.bin`中被调用，其源程序是`shell_0.c`

- [x] ~~void exit(int status);~~

  运行`test_1.bin`可调用，其源程序是`test_1.c`。

- [x] ~~int signal(int sig, void* handler, void* _Handler);~~

  运行`sig_0.bin`可调用，其源程序是`sig_0.c`。

- [ ] int sigsend(int pid, Sigaction* sigaction_p);
- [ ] void sigreturn(int ebp);
- [x] ~~u32 total_mem_size();~~

  运行`test_0.bin`可调用，其源程序是`test_0.c`。

- [x] ~~int shmget(int key, int size, int shmflg);~~

  运行`test_4.bin`可调用，其源程序是`test_4.c`。

- [x] ~~void* _shmat(int shmid, char* shmaddr, int shmflg);~~

  运行`test_4.bin`可调用，其源程序是`test_4.c`。

- [x] ~~void _shmdt(char* shmaddr);~~

  运行`test_4.bin`可调用，其源程序是`test_4.c`。

- [x] ~~struct ipc_shm* shmctl(int shmid, int cmd, struct ipc_shm* buf);~~

  运行`test_4.bin`可调用，其源程序是`test_4.c`。

- [ ] void* shmmemcpy(void* dst, const void* src, long unsigned int len);
- [x] ~~int ftok(char* f,int key);~~

  运行`test_5.bin`可调用，其源程序是`test_5.c`。

- [x] ~~int msgget(key_t key, int msgflg);~~

  运行`test_5.bin`可调用，其源程序是`test_5.c`。

- [x] ~~int msgsnd(int msqid, const void* msgp, int msgsz, int msgflg);~~

  运行`test_5.bin`可调用，其源程序是`test_5.c`。

- [x] ~~int msgrcv(int msqid, void* msgp, int msgsz, long msgtyp, int msgflg);~~

  运行`test_5.bin`可调用，其源程序是`test_5.c`。

- [x] ~~int msgctl(int msgqid, int cmd, msqid_ds* buf);~~

  注释掉`test_5.c`当前的`main`函数，取消第一个`main`函数的注释，运行`test_5.bin`可调用。

- [x] ~~void test(int no);~~

  注释掉`test_2.c`当前的`main`函数，取消”Syscall Pthread Test“下`main`函数的注释，运行`test_2.bin`可调用。

- [ ] u32 execvp(char* file, char* argv[]);
- [x] ~~u32 execv(char* path, char* argv[]);~~

  运行`shell_1.bin`可调用，，其源程序是`shell_1.c`但暂时有问题。

- [x] ~~pthread_t pthread_self();~~

  运行`ptest13.bin`可调用，其源程序是`fortest13.c`。
- [x] ~~int pthread_mutex_init(pthread_mutex_t* mutex, pthread_mutexattr_t* mutexattr);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~int pthread_mutex_destroy(pthread_mutex_t* mutex);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~int pthread_mutex_lock(pthread_mutex_t* mutex);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~int pthread_mutex_unlock(pthread_mutex_t* mutex);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~int pthread_mutex_trylock(pthread_mutex_t* mutex);~~

  运行`ptest4.bin`可调用，其源程序是`fortest4.c`。
- [x] ~~int pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* cond_attr);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex);~~

  运行`ptest13.bin`可调用，其源程序是`fortest13.c`。
- [x] ~~int pthread_cond_timewait(pthread_cond_t* cond, pthread_mutex_t* mutex, int timeout);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~int pthread_cond_signal(pthread_cond_t* cond);~~

  运行`ptest13.bin`可调用，其源程序是`fortest13.c`。
- [x] ~~int pthread_cond_broadcast(pthread_cond_t* cond);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [x] ~~int pthread_cond_destroy(pthread_cond_t* cond);~~

  运行`ptest10.bin`可调用，其源程序是`fortest10.c`。
- [ ] int get_pid_byname(char* name);
- [x] ~~int mount(const char *source, const char *target,const char *filesystemtype, unsigned long mountflags, const void *data);~~

  在`ktest.c`中有调用。
- [x] ~~int umount(const char *target);~~

  运行`test_0.bin`可调用，其源程序是`test_0.c`。
- [x] ~~int init_block_dev(int drive);~~

  在`ktest.c`中有调用。

五、如何添加一个新的系统调用

1. 在`user/include/syscall.h`中添加系统调用接口声明。例如：

   ```
   int read(int fd, void* buf, int count);
   ```

2. 在`user/include/syscall.h`中添加系统调用号。例如：

```
#define _NR_read      13
```

3. 在`user/ulib/syscall.c`中添加系统调用接口实现。例如：

```
// 说明：由于 read 需要三个参数，所以需要使用 _syscall3()
// 注：read 的系统调用号 _NR_read 不计入参数数量的统计中

int read(int fd, void* buf, int count) {
  return _syscall3(_NR_read, fd, buf, count);
}
```

4. 在想要添加新的系统调用实现的文件中以 sys_xx() → do_xx() → kern_xx() 的方式进行实现。例如：

```
PUBLIC int sys_read()
{
    return do_read(get_arg(1), get_arg(2), get_arg(3));
}

PUBLIC int do_read(int fd, char *buf, int count) {
    return kern_read(fd, buf, count);
}

PUBLIC int kern_read(int fd, char *buf, int count) {
    // Some operation.
}
```

5. 在`os/include/proto.h`和`os/kernel/syscall.c`中进行与 1 和 2 中同样的声明和实现。
6. 分别修改 os 目录和 user 目录下的 Makefile 文件

六、存在的问题

1. 在系统设计上，我们包含用户级进程、系统级进程和内核级进程三种。理论上来说，系统调用是提供给用户进程使用的，但 mini-os 的系统级进程（如 initial）也会调用系统调用接口，这使我们不得不在 os 目录下也保留一份系统调用接口，并在`proto.h`中进行声明。
2. 系统的结构不是很清晰。例如，任一对除本文件外的其他文件提供服务的函数应该在`proto.h`文件中进行声明，但事实上，一个函数在多个地方进行了声明。如`stdio.h`，该头文件的意义与proto.h有重合的地方，故已被我注释。