Università della Svizzera italiana

Faculty of Informatics

Operating Systems

PintOS: Report Project 6

Group 11

Giorgia Lillo, Lamberto Ragnolini

Fall Semester, May 21, 2024

1 Files changed

- /pintos-env/pintos/userprog/syscall.c

2 Changes

/pintos-env/pintos/userprog/syscall.c

```c
/* prototypes for the functions */
static void syscall_handler(struct intr_frame *);

static void syscall_exit (uint32_t *arguments, uint32_t *eax);
static void syscall_wait (uint32_t *arguments, uint32_t *eax);
static void syscall_exec (uint32_t *arguments, uint32_t *eax);

//added
static void syscall_write (struct intr_frame *f);
static void syscall_create (struct intr_frame *f);
static void syscall_remove(struct intr_frame *f);
static void syscall_open (struct intr_frame *f);
static void syscall_close (struct intr_frame *f);
static void syscall_read (struct intr_frame *f);
static void syscall_seek (struct intr_frame *f);
static void syscall_tell (struct intr_frame *f);
static void syscall_filesize (struct intr_frame *f);
static void syscall_halt (struct intr_frame *f);

typedef void (*handler) (uint32_t *, uint32_t *);
typedef void (*handler_frame) (struct intr_frame *);

void exit_with_status(int status);
static bool validate_arguments (uint32_t *arguments, int num_arguments);
void validate_addresses (void *ptr, uint32_t *eax, int num, int size);
```

Prototypes of the implemented functions.

```c
#define BAD_STATUS -1
#define SYSCALL_MAX_CODE 19
static handler call[SYSCALL_MAX_CODE + 1];

//added
static handler_frame call_frame[SYSCALL_MAX_CODE + 1];
```

New handler implemented that will be used for all system call numbers except for SYS_EXIT, SYS_EXEC and SYS_WAIT. The new handler will pass to the appropriate function directly the frame itself.

```c
* declare locks and fyles */
struct hash fd_table;
struct lock filesys_lock;
/* File descriptors numbered 0 and 1 are reserved for the console, documentation
     says*/
int next_fd = 2;

/* we need a struct to handle the the fd's for the specific items */
struct file_with_fd {
  struct file * file;
  struct hash_elem elem;
  int fd;
  char * thread_name;
};


static unsigned item_hash (const struct hash_elem* e, void* aux) {
  struct file_with_fd* i = hash_entry(e, struct file_with_fd, elem);
  return hash_int(i->fd);
}

static bool item_compare(const struct hash_elem* a, const struct hash_elem* b,
    void* aux) {
  struct file_with_fd *i_a = hash_entry(a, struct file_with_fd, elem);
  struct file_with_fd *i_b = hash_entry(b, struct file_with_fd, elem);
  return i_a->fd < i_b->fd;
}
```

This is just the declaration of the hash table and what is needed for it which is already explained in the slides. There is also the struct file_with_fd that maps appropriate file with its appropriate fd. The variable next_fd will be used to give appropriate unique fd's to the files, its initial value is 2 because numbers 0 and 1 are reserved for the console. There is also the declaration of the lock that will be used all over the handling of the system call to provide synchronization.

- syscall_init (*modified*)

NOT MODIFIED

```c
void syscall_init(void)
{
  intr_register_int(0x30, 3, INTR_ON, syscall_handler, "syscall");

  memset(call, 0, SYSCALL_MAX_CODE + 1);
```

ADDED

```c
  memset(call_frame, 0, SYSCALL_MAX_CODE + 1);


  call_frame[SYS_WRITE] = syscall_write;
  call_frame[SYS_CREATE] = syscall_create;
  call_frame[SYS_REMOVE] = syscall_remove;
  call_frame[SYS_OPEN] = syscall_open;
  call_frame[SYS_CLOSE] = syscall_close;
  call_frame[SYS_READ] = syscall_read;
  call_frame[SYS_SEEK] = syscall_seek;
  call_frame[SYS_TELL] = syscall_tell;
  call_frame[SYS_FILESIZE] = syscall_filesize;
  call_frame[SYS_HALT] = syscall_halt;
```

Just simple initialization of the call frame handler and mapping of appropriate functions to system call numbers.

NOT MODIFIED

```c
  /* initialize the handler providing the function to handle, in our case
     only write and exit*/
  call[SYS_EXIT] = syscall_exit;
  call[SYS_WAIT]  = syscall_wait;
  call[SYS_EXEC]  = syscall_exec;
```

ADDED

```c
  hash_init(&fd_table, item_hash, item_compare, NULL);
  lock_init(&filesys_lock);
}
```

Just simple initialization of the hash table and the lock.

- syscall_handler (*modified*)

NOT MODIFIED

```c
static void
syscall_handler(struct intr_frame *f)
{
  /* if f is not a valid pointer then just delegate to the approriate helper
       function to exit with appropriate exit status */
  if (f == NULL)
    exit_with_status(BAD_STATUS);
  /* we check if the first argument on the stack is valid with esp which
     points at the top of the stack, this will be the system call number */
  uint32_t* arguments = ((uint32_t*) f->esp);
  // this fixes the wait and sc-bad-sp tests
  if(!validate_arguments(arguments, 1)) exit_with_status(BAD_STATUS);
```

ADDED

```
  if(*arguments == SYS_EXIT || *arguments == SYS_EXEC || *arguments ==
     SYS_WAIT) {
    call[*arguments](++arguments, &(f->eax));
  } else {
    call_frame[*arguments](f);
  }
}
```

If the system call numbers are either SYS_EXIT, SYS_EXEC and SYS_WAIT, we delegate to old implementation with the normal handler, otherwise for the other system call numbers we use the new implemented handler and we pass directly the frame itself.

- syscall_create (*added*)

```

static void syscall_create (struct intr_frame *f) {

  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  const char * file = stack[1];
  unsigned initial_size = stack[2];

  /* check params */
  if (!file || !is_user_vaddr(file)) {
      *eax = false;
      exit_with_status(BAD_STATUS);
      return;
  }

  if(pagedir_get_page (thread_current ()->pagedir, file) == NULL ) {
    *eax = false;
    exit_with_status(BAD_STATUS);
    return;
  }

  if (initial_size < 0) {
      *eax = false;
      return;
  }

  /* Acquire lock to ensure synchronization */
  lock_acquire(&filesys_lock);
  *eax = filesys_create(file, initial_size);
  lock_release(&filesys_lock);
}
```

After retrieving the parameters on the stack appropriate checks with respect to tests needs to be done. We know that the user may provide an invalid pointer which would mean:
- a null pointer

- a pointer to kernel address space
- a pointer to unmapped virtual memory.

We then provide appropriate checks for them. Based on the tests, other than this we have to check also other minor things such as the size. After the checks we provide synchronization with the lock initialized before and we call in the critical section the appropriate file system function.

- syscall_remove (*added*)

```c

static void
syscall_remove (struct intr_frame *f){

  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  const char * file = stack[1];

  /* check params */
  if (!file || !is_user_vaddr(file)) {
      *eax = false;
      return;
  }

  lock_acquire (&filesys_lock);
  *eax = filesys_remove(file);
  lock_release (&filesys_lock);
}
```

After retrieving the parameters on the stack appropriate checks with respect to tests needs to be done. We know that the user may provide an invalid pointer which would mean:
- a null pointer
- a pointer to kernel address space
- a pointer to unmapped virtual memory.

We then provide appropriate checks for them. After the checks we provide synchronization with the lock initialized before and we call in the critical section the appropriate file system function.

- syscall_open (*added*)

```c
static void
syscall_open (struct intr_frame *f){

  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  const char * file = stack[1];

  /* check params */
  if (!file || !is_user_vaddr(file)) {
      *eax = false;
      exit_with_status(BAD_STATUS);
      return;
  }


  if (pagedir_get_page (thread_current ()->pagedir, file) == NULL) {
    *eax = false;
    exit_with_status(BAD_STATUS);
    return;
  }

  if(strlen(file) < 1){
    *eax = -1;
    return;
  }

  /* allocate memory for the struct */
  struct file_with_fd * allocated_fd = malloc(sizeof(struct file_with_fd));
  /* check that allocation didn't fail*/
  if (!allocated_fd) {
    *eax = -1;
    return;
  }
  /* acquire lock so that you are sure that you enable synchronization */
  lock_acquire (&filesys_lock);

  struct file* fopen;
  fopen = filesys_open(file);
  if (!fopen) {
    free(allocated_fd);
    lock_release (&filesys_lock);
    *eax = -1;
    return;
  }
  /* fill the struct and insert maping in hash, return file descriptor */
  allocated_fd->file = fopen;
  allocated_fd->fd = next_fd++;
  allocated_fd->thread_name = thread_current()->name;
  hash_insert(&fd_table, &allocated_fd->elem);
  *eax = allocated_fd->fd;
  lock_release (&filesys_lock);
}
```

After retrieving the parameters on the stack appropriate checks with respect to tests needs to be done. We know that the user may provide an invalid pointer which would mean:
- a null pointer
- a pointer to kernel address space
- a pointer to unmapped virtual memory.
We then provide appropriate checks for them. Based on the tests we need to check other things such as if the file is empty. After the checks we provide synchronization with the lock initialized before and we call in the critical section the appropriate file system function. If the file system function succedes we allocate memory for the file_with_fd struct where we will save a unique fd together with the opened file. We then insert it into the hash table. If instead it fails we deallocate memory, release the lock and return -1 as error.

- syscall_close (*added*)

```
static void
syscall_close(struct intr_frame *f){

   /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  int fd = stack[1];

  lock_acquire (&filesys_lock);
  /* if there is no entry in the hash table do nothing and release lock*/
  struct file_with_fd closing_file;
  closing_file.fd = fd;
  struct hash_elem* helem = hash_find(&fd_table, &closing_file.elem);
  if (helem) {
    struct file_with_fd* file_to_close = hash_entry(helem, struct
        file_with_fd, elem);
    struct file * ff = file_to_close->file;
     if(strcmp(thread_current()->name, file_to_close->thread_name) == 0) {
      file_close(ff);
      hash_delete(&fd_table, &(file_to_close->elem));
      free(file_to_close);
      }
    }
  lock_release (&filesys_lock);
}
```

We initially retrieve the fd from the stack, after this we provide synchronization with the lock initialized before and in the critical section we search the file with the fd passed on the stack in hash table with the appropriate methods already explained in the slides, if the file is in the hash table and the current thread is the one that opened it then we close it with the appropriate file system function and we delete the mapping in the hash table.

- syscall_filesize (*added*)

```
tatic void
syscall_filesize(struct intr_frame *f){

  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  int fd = stack[1];

  lock_acquire (&filesys_lock);
  struct file_with_fd search_fd;
  search_fd.fd = fd;
  struct hash_elem* helem = hash_find(&fd_table, &search_fd.elem);
  if (!helem) {
    *eax = 0;
    lock_release(&filesys_lock);
    return;
  }
  /* otherwise find it in the hash table and then returns its size*/
  struct file_with_fd* file_for_size = hash_entry(helem, struct file_with_fd
      , elem);
  if(!file_for_size) {
    *eax = 0;
    lock_release (&filesys_lock);
    return;
  }
  /* return correct value */
  *eax = file_length(file_for_size->file);
  lock_release (&filesys_lock);

}
```

We initially retrieve the fd from the stack, after this we provide synchronization with the lock initialized before and in the critical section we search the file with the fd passed on the stack in hash table with the appropriate methods already explained in the slides, if the file is in the hash table we return its length by means of the appropriate file system function. If instead the file is not in the hash table we will release the lock and return 0.

- syscall_read (*added*)

```
static void
syscall_read(struct intr_frame *f){

  lock_acquire (&filesys_lock);

  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  int fd = stack[1];
  void * buffer = stack[2];
  unsigned size = stack[3];

  /* check params */
  if (!buffer || !is_user_vaddr(buffer)) {
      *eax = -1;
      exit_with_status(BAD_STATUS);
      return;
  }

  if(pagedir_get_page (thread_current ()->pagedir, buffer) == NULL ) {
    *eax = -1;
    exit_with_status(BAD_STATUS);
    return;
  }

  /* use return call function value initiliazied to -1*/
  int ret = -1;
  if(fd==0) {
    int i;
    for (i = 0; i < size; i++) {
      if (input_getc() == '\0') break;
    }
    ret = i;
  } else {
    /* retrieve the file from the map */
    struct file_with_fd reading_file;
    reading_file.fd = fd;
    struct hash_elem* helem = hash_find(&fd_table, &reading_file.elem);
    if (!helem){
      *eax = ret;
      lock_release (&filesys_lock);
      return;
    }
    /* if it exists retrieve it and read it and save in return call variable
       the num of bytes read */
    struct file_with_fd * file_to_read = hash_entry(helem, struct
       file_with_fd, elem);
    if (file_to_read && file_to_read->file) {
      ret = file_read(file_to_read->file, buffer, size);
    }
    /*if it will fail/does not exist then ret is already -1 */
  }
  *eax = ret;
  lock_release (&filesys_lock);

}
```

After retrieving the parameters on the stack appropriate checks with respect to tests needs to be done. We know that the user may provide an invalid pointer which would mean:
- a null pointer
- a pointer to kernel address space
- a pointer to unmapped virtual memory.
We then provide appropriate checks for them. Synchronization is provided with the lock initialized before, we then check if the fd passed on the stack is 0 it means that we'll need to read from the keyboard, so we use the appropriate function to do it. If instead the fd is not 0 we will search into the hash table for the file for the file with the fd passed on the stack and if it exists we'll use the appropriate file system function to read into buffer and we will return how many bytes have been read into buffer. If the file instead is not in the hash table we will release the lock and return -1.

- syscall_write (*modified*)

```
static void
syscall_write(struct intr_frame *f){

  lock_acquire (&filesys_lock);
  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  int fd = stack[1];
  const void * buffer = stack[2];
  unsigned size = stack[3];

    /* check params */
  if (!buffer || !is_user_vaddr(buffer)) {
      *eax = -1;
      exit_with_status(BAD_STATUS);
      return;
  }

  if(pagedir_get_page (thread_current ()->pagedir, buffer) == NULL ) {
    *eax = -1;
    lock_release(&filesys_lock);
    exit_with_status(BAD_STATUS);
    return;
  }

  if(fd == 1) {
    /* use appropriate function to print to stdout*/
    putbuf(buffer, size);
    /* save important stuff in this case the length of the buffer to eax*/
    *eax = size;
  } else {
    /* retrieve the file to write */
    struct file_with_fd file_to_write;
    file_to_write.fd = fd;
    struct hash_elem* helem = hash_find(&fd_table, &file_to_write.elem);
    if (helem) {
      struct file_with_fd* file_in_writing = hash_entry(helem, struct
          file_with_fd, elem);
      struct file * ff = file_in_writing->file;
      *eax = file_write(ff, buffer, size);
    }
  }
  lock_release (&filesys_lock);
}
```

After retrieving the parameters on the stack appropriate checks with respect to tests needs to be done. We know that the user may provide an invalid pointer which would mean:
- a null pointer
- a pointer to kernel address space
- a pointer to unmapped virtual memory.
We then provide appropriate checks for them. Synchronization is provided with the lock initialized before, we then check if the fd passed on the stack is 1 it means that we'll need to write to the console, this was already implemented with previous project. If instead the fd is not 1 we will search into the hash table for the file for the file with the fd passed on the stack and if it exists we'll use the appropriate file system function to write from the buffer into the file and we will return how many bytes have been written which may be less than size if some bytes could not be written. If the file instead is not in the hash table we will simply just release the lock.

- syscall_seek (*modified*)

```
static void
syscall_seek(struct intr_frame *f){

  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  int fd = stack[1];
  unsigned position = stack[2];

  lock_acquire (&filesys_lock);
  /* try to retrieve the file if it doens't exist then release the lock*/
  struct file_with_fd seeked_file;
  seeked_file.fd = fd;
  struct hash_elem* helem = hash_find(&fd_table, &seeked_file.elem);
  if (!helem) {
    lock_release (&filesys_lock);
    return;
  }
  /* otherwise if it exists retrieve the file */
  struct file_with_fd* file_to_seek = hash_entry(helem, struct file_with_fd,
      elem);
  if(file_to_seek && file_to_seek->file) {
    file_seek(file_to_seek->file, position);
  }
  lock_release (&filesys_lock);

}
```

We initially retrieve the fd and the position from the stack, after this we provide synchronization with the lock initialized before and in the critical section we search the file with the fd passed on the stack in hash table with the appropriate methods already explained in the slides, if the file is in the hash table we call appropriate file system function. If instead the file is not in the hash table we will just release the lock.

- syscall_tell (*modified*)

```
static void
syscall_tell(struct intr_frame *f){

  /* retrieve params */
  uint32_t* eax = &(f->eax);
  unsigned int * stack = f->esp;
  int fd = stack[0];

  lock_acquire (&filesys_lock);
  unsigned int pos = 0;
  struct file_with_fd telled_file;
  telled_file.fd = fd;
  struct hash_elem* helem = hash_find(&fd_table, &telled_file.elem);
  if (!helem) {
    *eax = pos;
    lock_release(&filesys_lock);
    return;
  }
  /* otherwise actually retrieve the file */
  struct file_with_fd* file_to_tell = hash_entry(helem, struct file_with_fd,
      elem);
  if(file_to_tell && file_to_tell->file) {
    pos = file_tell(file_to_tell->file);
  }
  *eax = pos;
  lock_release (&filesys_lock);
}
```

We initially retrieve the fd from the stack, after this we provide synchronization with the lock initialized before and in the critical section we search the file with the fd passed on the stack in hash table with the appropriate methods already explained in the slides, if the file is in the hash table we call appropriate file system function and return the position of the next byte to read. If instead the file is not in the hash table we will return position 0.

- syscall_halt (*modified*)

```
/*halt system call*/
static void
syscall_halt(struct intr_frame *f){
   shutdown_power_off();
}
```

Nothing to explain here.