Università della Svizzera italiana

Faculty of Informatics

Operating Systems

PintOS: Report Project 5

Group 11

Giorgia Lillo, Lamberto Ragnolini

Fall Semester, May 6, 2024

1 Files changed

- /pintos-env/pintos/userprog/syscall.c
- /pintos-env/pintos/threads/thread.c
- /pintos-env/pintos/userprog/process.c

2 Changes

/pintos-env/pintos/userprog/syscall.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_write (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);
typedef void (*handler) (uint32_t *, uint32_t *);
void exit_with_status(int status);
static bool validate_arguments (uint32_t *arguments, int num_arguments);
static bool safe_check (void *argument);

//added
#define BAD_STATUS -1
```

The BAD_STATUS macro is used every time we have an exit with a failure/error, or a behaviour that is not the right one. As written in the slides and in the assignments every time something like

this happens the exit status code should be -1, so we refactor it into this macro.

- syscall_init (*modified*)

  NO MODIFIED

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

  memset(call, 0, SYSCALL_MAX_CODE + 1);

  /* initialize the handler providing the function to handle, in our case
     only write and exit*/
  call[SYS_EXIT] = syscall_exit;
  call[SYS_WRITE] = syscall_write;
```

  MODIFIED

```
  //added
  call[SYS_WAIT]  = syscall_wait;
  call[SYS_EXEC]  = syscall_exec;
}
```

  Simply added the mapping of the syscall number with its appropriate function, this was already explained in the last assignment, nothing new, just the handling of the new implemented system calls.

- syscall_handler (*modified*)

```
static void
syscall_handler(struct intr_frame *f)
{
  // added
  /* 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);
  int arg_to_check_num = 1;
  /* we check if the argument is valid by delegating to the appropriate
     helper function */
  if (!validate_arguments (arguments, arg_to_check_num))
    exit_with_status(BAD_STATUS);

  /* retrieve sys call number which is what is arguments pointing to and
     enter the call handler with it*/
  /* now it would be useless to pass the sys call number to the handler so
     we make arguments point to the actual first argument by incrementing
     the pointer */
  call[*arguments](++arguments, &(f->eax));
}
```

Changes are made to this function in order to handle the system calls in a better way. As asked in the slide we need to check that all the pointers are valid, so initially we check if the frame pointer which is the one passed in this function as argument is a valid one, so it is not null, in case it is null we delegate to the proper function that handles the exit with the bad status code of -1. Soon after we retrieve the argument standing on top of the stack which will be the system call number, we check if it is valid, if so we use the call handler to call the appropriate mapped function, otherwise we delegate to the helper exit function with the bad status code -1.

- syscall_exit (*modified*)

```
/* function that handles the exit system call*/
// changed argument passing
static void
syscall_exit(uint32_t *arguments, uint32_t *eax)
{
  //added
  /* if we are in this system call then we know that the arguments on the
     stack being pushed is only one*/
  int arg_on_stack = 1;
  /* initialize a status where we will save the status code*/
  int status;
  /* we check if the argument is valid by delegating to the appropriate
     helper function */
  if (validate_arguments(arguments, arg_on_stack))
    /* if the argument is valid the we retrieve it, we know that the exit
       system call puts on the stack just an
       argument (watch syscall.c in /lib/user) which is a status code so we
          retrieve it */
    status = (int) *arguments;
  else
    /* if the argument is not valid then place the BAD_STATUS code in the
       status */
    status = BAD_STATUS;

  /* after that we delegate to the appropriate functio to have an
     appropriate exit */
  exit_with_status(status);
}
```

```
/* function that handles the write system call*/
// changed the argument passing
static void
syscall_write(uint32_t *arguments, uint32_t *eax)
{
  // added
  /* if we are in this system call then we know that the arguments on the
     stack being pushed are three */
  int arg_on_stack = 3;
  /* delegate to helper functions the check for the validity of the
     arguments, we also check the address of the buffer to see if its a
     safe one*/
  if (!validate_arguments(arguments, arg_on_stack) || (pagedir_get_page(
     thread_current()->pagedir, arguments[1]) == NULL))
    /* in case the validity of the arguments or the buffer address is not
       safe we need to exit with BAD_STATUS as status*/
```

```
    exit_with_status(BAD_STATUS);

  //added
  /* now we need to pop the arguments from the stack thatthe syscall pushed
     to retrieve the fd, the buffer and its len*/
  /*firstwe retrieve the fd which should be 1 since we are in a write
     syscall handling*/
  int fd = (int) arguments[0];
  /* once retrieved the fd we assure that it is 1*/
  ASSERT(fd == 1);
  /* the we retrieve the actual buffer */
  char *buffer = (char *) arguments[1];
  /*then we retrieve the length of the buffer*/
  int len = (int) arguments[2];

  //added
  /* use appropriate function to print to stdout*/
  putbuf(buffer, len);
  /* save important stuff in this case the length of the buffer to eax*/
  *eax = len;
}
```

Given the changes applied to the argument passing of the handler we had to adjust also the old implemented system calls, which are syscall exit and syscall write. Given that they are not required in this assignment and are already explained in the last assignment we will not explain the changes but we've provided detailed comment section.

- syscall_wait (*added*)

```
//added
/* function that handles the wait system calls*/
static void
syscall_wait (uint32_t *arguments, uint32_t *eax) {

  /* if we are in this system call then we know that the arguments on the
     stack being pushed is only one (watch syscall.c in /lib/user)*/
  int arg_on_stack = 1;

  /* check that arguments are valid by delegating to helper function*/
  if (!validate_arguments(arguments, arg_on_stack))
    /* if they are not valid exit with status BAD_STATUS*/
    exit_with_status(BAD_STATUS);

  /*  if arguments are valid just call appropriate function and save the
     output into eax*/
  uint32_t result = process_wait((int) *arguments);
  *eax = result;
}
```

This function handles the wait system calls if we are in this function we already know that the wait has pushed only one argument on the stack which is the pid of the child. So afterwords we validate the appropriate function the argument passed as a parameter to this function, in case the parameter is not valid we delegate the appropriate exit function with bad status of -1 otherwise we call the appropriate process function which is process wait, passing the argument to it. We store the output of process wait into eax.

- syscall_exec (*added*)

```
//added
/* function that handles the exec system calls*/
static void
syscall_exec (uint32_t *arguments, uint32_t *eax) {

  /* if we are in this system call then we know that the arguments on the
     stack being pushed is only one (watch syscall.c in /lib/user)*/
  int arg_on_stack = 1;
  /* check validity of arguments and pointers as requested*/
  if (!validate_arguments(arguments, arg_on_stack) || (pagedir_get_page(
     thread_current()->pagedir, arguments[1]) == NULL))
    /*exit with status BAD_STATUS in case not*/
    exit_with_status(BAD_STATUS);

  /*delegate to approriate function that will handle the execution and save
     output to eax*/
  uint32_t result = process_execute((char *) *arguments);
  *eax = result;
}
```

This function handles the exec system call, we know that the exec system call puts just one parameter on the stack. We need to check as requested the validity of the argument and its pointer and address, so we delegate to the helper function which will be explained later. We also need to check if the UADDR is unmapped with specific function. In case argument-pointer-address is not valid we delegate to appropriate exit function with bad status of -1. In case everything is fine we will call the function process execute which will handle the execution and we will save the return output into eax.

- validate_arguments (*added*)

```
//added
static bool
validate_arguments (uint32_t *arguments, int num_arguments)
{

  /* The user may provide an invalid pointer in a syscall
      - a null pointer
      - a pointer to kernel address space
      - a pointer to unmapped virtual memory
     We should control this*/

  /* we then need to check that what the user passed is ok and not whats in
     the list above */
  int i;
  for (i = 0; i < num_arguments + 1; i++, arguments++) {
    /* is_user_vadr : Returns true if VADDR is a user virtual address.
       pagedir_get_page : Looks up the physical address that corresponds to
          user virtual
                          address UADDR in PD.  Returns the kernel virtual
                             address
                          corresponding to that physical address, or a null
                             pointer if UADDR is unmapped.
     we also check if the pointer is null  */
```

```
    if (!(is_user_vaddr(arguments) && pagedir_get_page(thread_current()->
        pagedir, arguments) != NULL && arguments != NULL)) {
      return false;
    }
  }
  return true;
}
```

This is a helper function that validates the arguments passed form the system calls. Every time there is a handling of a system call we need to check that all the arguments passed are valid, this means that there should not be null pointers, pointer to kernel address space, pointer to unmapped virtual memory. In order to do this we use appropriate function already defined to check kernel address space and unmapped virtual memory and a simple check to check that the pointers are not null. In case all of these conditions are satisfied the function will return true, otherwise if the validity is not satisfied the function will return false.

- exit_with_status (*added*)

```
//added
void
exit_with_status (int status)
{
  /*retrieve the current thread and set to it the status that we passed as
     argument to the function*/
  struct thread * t = thread_current();
  t->exit_status = status;
  /*print the exit status of the thread*/
  printf("%s: exit(%d)\n", thread_current ()->name, status);
  /*after this delegate to thread_exit method to finish exit */
  thread_exit ();
}
```

This function is a helper function for the exit handling. It is called every time we need a thread to exit, it retrieves the current thread and sets the status passed as parameter to the exit status field in the thread. Soon after we print the exit status of the thread with its correspective name and after all of this we delegate the exit to the thread_exit method.

/pintos-env/pintos/threads/thread.c

- thread_schedule_tail (*modified*)

NOT MODIFIED

```
void thread_schedule_tail(struct thread *prev)
{
  struct thread *cur = running_thread();

  ASSERT(intr_get_level() == INTR_OFF);

  /* Mark us as running. */
  cur->status = THREAD_RUNNING;

  /* Start new time slice. */
  thread_ticks = 0;
```

```
12
13    #ifdef USERPROG
14      /* Activate the new address space. */
15      process_activate();
16    #endif
17
18    /* If the thread we switched from is dying, destroy its struct
19       thread.  This must happen late so that thread_exit() doesn't
20       pull out the rug under itself.  (We don't free
21       initial_thread because its memory was not obtained via
22       palloc().) */
23    if (prev != NULL && prev->status == THREAD_DYING && prev != initial_thread
         )
24    {
25      ASSERT(prev != cur);
```

MODIFIED

```
1       //added
2       /* in order to handle the correct execution of the wait we need to
           handle the right freeing of the pages for the treads
3          so we need to add this check when we are in USERPROG*/
4       #ifdef USERPROG
5         if (!(prev->pwait))
6           palloc_free_page(prev);
7       #else
8           palloc_free_page(prev);
9       #endif
10    }
11  }
```

In order to provide a correct execution of the wait if USERPROG is defined we have to provide an additional check on whether the parent of the diying thread is waiting. We will free the page only if the parent is not waiting.

/pintos-env/pintos/userprog/process.c

- process_execute (*modified*)

NOT MODIFIED

```
1   tid_t process_execute(const char *file_name)
2   {
3     char *fn_copy;
4
5     tid_t tid;
6
7     /* Make a copy of FILE_NAME.
8        Otherwise there's a race between the caller and load(). */
9     fn_copy = palloc_get_page(0);
10    if (fn_copy == NULL)
11      return TID_ERROR;
12    strlcpy(fn_copy, file_name, PGSIZE);
```

MODIFIED

```
1
2   //added
3   char *fn_other_copy;
4   char *save_ptr;
5   /* we make a second copy so we can check if the executable is valid or not
6      palloc_get_page : Obtains a single free page and returns its kernel
           virtual address.
7                        If PAL_USER is set, the page is obtained from the user
                             pool,
8                        otherwise from the kernel pool.  If PAL_ZERO is set in
                             FLAGS,
9                        then the page is filled with zeros.  If no pages are
10                       available, returns a null pointer, unless PAL_ASSERT
                            is set in
11                       FLAGS, in which case the kernel panics.*/
12  fn_other_copy = palloc_get_page (0);
13  if (fn_other_copy == NULL)
14    return TID_ERROR;
15  strlcpy (fn_other_copy, fn_copy, PGSIZE);
16
17  // added
18  /* Now we check the actual validity of the executable by analyzing the
       first argument of the command line*/
19  char *cmd_first = strtok_r(fn_other_copy, " ", &save_ptr);
20  if (!filesys_open(cmd_first)) {
21    /* in case the validity is not satisfied we will free the allocated
         pages and return a proper error*/
22    palloc_free_page(fn_copy);
23    palloc_free_page(fn_other_copy);
24    return TID_ERROR;
25  }
```

In this function we need to handle the possibility that the parameter passed to process_execute which is the command line is actually a valid command. In order to do this we need to have a second copy of the command line, so that we are able to use strtok_r function to retrieve the first argument of the command line and check if it is a valid executable. The need to create a second copy is due to the fact that strtok_r function actually trims the string so for the check we need a copy in order not to corrupt the actual string.

NOT MODIFIED

```
1   /* Create a new thread to execute FILE_NAME. */
2   tid = thread_create(file_name, PRI_DEFAULT, start_process, fn_copy);
3   if (tid == TID_ERROR)
4     palloc_free_page(fn_copy);
5   return tid;
6 }
```

- process_exit (*modified*)

NOT MODIFIED

```
1 /* Free the current process's resources. */
2 void process_exit(void)
3 {
4   struct thread *cur = thread_current();
```

```
  uint32_t *pd;

  /* Destroy the current process's page directory and switch back
     to the kernel-only page directory. */
  pd = cur->pagedir;
  if (pd != NULL)
  {
    /* Correct ordering here is crucial.  We must set
       cur->pagedir to NULL before switching page directories,
       so that a timer interrupt can't switch back to the
       process page directory.  We must activate the base page
       directory before destroying the process's page
       directory, or our active page directory will be one
       that's been freed (and cleared). */
    cur->pagedir = NULL;
    pagedir_activate(NULL);
    pagedir_destroy(pd);
  }
```

MODIFIED

```
  /* In order for the tests to pass we neeed to print the exit statuss */
  //printf("%s: exit(%d)\n", cur->name, cur->exit_status);
```

Given that we have a function in syscall.c that handles the exit of a thread we will print there this line. So we comment it here.

NOT MODIFIED

```

  /* check if the parent called wait, if so then we need to unblock him */
  if (cur->pwait)
    thread_unblock(cur->parent);
}
```