Università della Svizzera italiana

Faculty of Informatics

Operating Systems

PintOS: Report Project 4

Group 11

Giorgia Lillo, Lamberto Ragnolini

Fall Semester, April 24, 2024

1 Files changed

- /pintos-env/pintos/userprog/syscall.c
- /pintos-env/pintos/threads/thread.h
- /pintos-env/pintos/threads/thread.c
- /pintos-env/pintos/lib/string.h
- /pintos-env/pintos/lib/string.c
- /pintos-env/pintos/userprog/process.c

2 Changes

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

```
1 /* declaring prototypes*/
2 typedef void (*handler)(struct intr_frame *);
3 static void syscall_exit(struct intr_frame *f);
4 static void syscall_write(struct intr_frame *f);
5
6 /*declaring call handler*/
7 #define SYSCALL_MAX_CODE 19
8 static handler call[SYSCALL_MAX_CODE + 1];
```

- syscall_init (*modified*)

```
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;
```

- syscall_handler (*modified*)

```
/*handle the calls requested that are the write and the exit with the
   handler */
  /*retrieve the system call number from the stack*/
  int syscall_num = *(int *)f->esp;
  /*call the appropriate function based on the sys call number*/
  call[syscall_num](f);
```

- syscall_exit (*added*)

```
/*f->esp points to the top of the stack so we point now to the top of the
   stack*/
  int *stk = f->esp;
  /* retrieve the current thread */
  struct thread *t = thread_current();
  /* retrieve the exit status from the stack and place it into the thread
     struct*/
  int *tmp = stk + 1;
  t->exit_status = *tmp;
  /* call the apropriate function to conclude the thread exit*/
  thread_exit();
```

- syscall_write (*added*)

```
 /*we need to go through the stack to retrive all the needed paramteres
    pushed before from the program*/
  int *stk = f->esp;
  /*retrieve the buffer*/
  int *tmp1 = stk + 2;
  char *buffer = *tmp1;
  /*retrieve the length of the buffer*/
  int *tmp2 = stk + 3;
  int len = *tmp2;
  /*use appropriate function to print on the stdout in one go the buffer
     context*/
  putbuf(buffer, len);
  /*save in the important information, in our case the length into the f->
     eax*/
  f->eax = len;
```

Given that most of the explanation is already in the comments section of the code, we basically described in general how it all together works. In the syscall_init function we initialize the

proper resources such as the call handler, mapping the right syscall number to the right syscall function. In the syscall_handler function we handle the proper calls by retrieving the syscall number from the stack and calling the proper function. The other two functions are the function that will handle the appropriate syscalls already explained in details in the comment code sections.

/pintos-env/pintos/threads/thread.h

- struct thread (*modified*)

```
#ifdef USERPROG
  /* Owned by userprog/process.c. */
    uint32_t * pagedir;                 /* Page directory. */
    int exit_status;                    /* Status*/
    struct thread* parent;              /* Parent thread */
    bool pwait;                /* true means parent is waiting,false means
       parent is not waiting */
#endif
```

Added needed variable resources to keep track of the parent thread, the parent waiting time and exit status. These are all variables that we need to keep track of the parent child relations and the exit status of a thread.

- struct thread * thread_get_by_tid (*added*)

  Declaration of the prototype function defined im thread.c and used in process.c file.

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

- struct thread * thread_get_by_tid (*added*)

```
struct thread *
thread_get_by_tid(int tid)
{
  /*iterate over the all_list to search for the thread with the same tid
     passed in function*/
  struct list_elem *it;
  for (it = list_begin(&all_list);
       it != list_end(&all_list);
       it = list_next(it))
  {
    /*check the match by tid*/
    struct thread *tt = list_entry(it, struct thread, allelem);
    if (tt->tid == tid)
    {
      /*the match has been successfull so we return the matched thread*/
      return tt;
    }
  }
}
```

This function returns the thread who's tid matches the requested one. A simple search by tid in the all_list.

- tid_t thread_create (*modified*)

```
#ifdef USERPROG
  /* thread default exit code is an error value in
     order to prevent a possible fail in the assignment of it */
  t->exit_status = TID_ERROR;
  /* assign parent to the newly created thread which will obviously be
     the current running thread */
  t->parent = thread_current();
  /* used to let the child klnow if the parent is waiting or not for the
      completion */
  t->pwait = false;
#endif
```

- static void init_thread (*modified*)

```
#ifdef USERPROG
  /* the thread name should be just the filename that it executes,
    excluding any additional arguments passed in command line */
  strcpyfw(t->name, name, 16);
#else
  /*otherwise just compute normal execution*/
  strlcpy(t->name, name, sizeof t->name);
#endif
```

No need for further explanation, all explained in the comments.

/pintos-env/pintos/lib/string.h

- void strcpyfw(char * dst, const char * src, size_t destbuflen)(*added*)

  Declaration of the prototype function defined in string.c file.

/pintos-env/pintos/lib/string.c

- void strcpyfw(*added*)

```
void
strcpyfw(char * destination, const char * source, size_t dstbufferlen) {
  strlcpy(destination, source, dstbufferlen);
  char * save;
  strtok_r(destination, " ", &save);
}
```

This function copies only the first part of the name because is the only one needed in USERPROG case, with the

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

```
/*arbitrarly choose values for the arguments and the filename length, macros
    are better than normal variables in this case, more C oriented*/
#define MAX_ARGUMENTS 32
#define FILE_LEN_MAX 255

/* prototypes */
void actual_push(void **, int, void *);
void push_on_stack(void **, char *);
```

- void actual_push*(added)*

```
/* This function is a helper function needed to actually copy the parameters
    onto the stack it decrease the top pointer and copies the parameter
   onto the stack, we need to remeber that the stack grows downwards */
void actual_push(void **top, int length, void *param)
{
  /* decrease the top pointer by the length of the param REMEMBER stack
     grows downwards */
  *top -= length;
  /* copy the parameter onto the stack*/
  memcpy(*top, param, length);
}
```

We added this helper function that is called in the helper function push_on_stack(). No need for further explanation, all explained in the comments.

- void push_on_stack*(added)*

```
/*this function puts all the needed parameters onto the stack from which
   then the OS is gonna pop*/
void push_on_stack(void **top, char *cmd)
{
  /* retrieve the base of the stack */
  void *base = *top;

  /* initialize the address array for the parameters where to store
     addresses */
  char *addressarr[MAX_ARGUMENTS];
  /* pointer for the single argument everytime */
  char *param;
  /* save pointer needed for the strtok_r function*/
  char *save;
  /* counter needed to count the number of parameters pushed (parameters on
     the command line)*/
  int argc = 0;
  /* push arguments on top of the stack by tokenizing them with the proper
     function, simple for loop that iterates over the command line and
     tokenizes the arguments in order to be able to  push them separately
     on the stack */
  for (param = strtok_r(cmd, " ", &save);
       param != NULL;
       param = strtok_r(NULL, " ", &save))
  {
    /* now push the tockenized parameter onto the stack*/
    /* calculate length of the parameter */
    int length = (strlen(param) + 1);
    /* the push in this way is going to be done multiple times so we
       refactor a helper function to do it*/
    actual_push(top,length, param);
    /* save the address where the current parameter was saved into its
       proper array*/
    addressarr[argc++] = *top;
  }

  /* as in the stack photo on the slides the last element of the address
     array is a 0 so we place it */
  addressarr[argc] = 0;
```

```
31
32    /* following the stack photo on the slides we need
33       the word-align to make it align which makes sense so we align it*/
34    int wrd_align = 4 - ((base - *top) % 4);
35    *top -= wrd_align;
36
37    /* pointer size needed for our helper function */
38    int psize = sizeof(void *);
39
40    /* now push argument's addresses onto the stack with a simple for loop
         that iterates
41       over the address array */
42    int i;
43    for (i = argc; i >= 0; i--)
44    {
45      actual_push(top, psize, &addressarr[i]);
46    }
47
48    /* now we need to put argv onto the stack as also in the slides
         representation */
49    char *argvarraddr = *top;
50    actual_push(top, psize, &argvarraddr);
51
52    /* we need to put argc onto the stack as also in the slides representation
          */
53    actual_push(top, psize, &argc);
54
55    /* now we need to put the return address onto the stack as also in the
         slides representation, the return address is a 0 so we first declare
         and initialize it*/
56    int return_addr = 0;
57    actual_push(top, psize,  &return_addr);
58  }
```

This is an helper function that we will call in the function start_process(), in order to loads a user process in a correct way. We basically push onto the stack all the needed parameters that are before tokenized with the proper function, moreover, at the end, we also push the argv, argc and the return address as represented in the slides.

- static void start_process(*modified)*

  MODIFIED:

```
1   /* A thread function that loads a user process and starts it
2      running. */
3   static void
4   start_process(void *file_name_)
5   {
6     /* file_name_ contains all the argument passed on command line with the
         file name included.
7        fname contains ONLY the name of the file instead*/
8     char fname[FILE_LEN_MAX];
9     strcpyfw(fname, file_name_, FILE_LEN_MAX);
10
11    struct intr_frame if_;
12    bool success;
```

NOT MODIFIED:

```
  /* Initialize interrupt frame and load executable. */
  memset(&if_, 0, sizeof if_);
  if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
  if_.cs = SEL_UCSEG;
  if_.eflags = FLAG_IF | FLAG_MBS;
```

MODIFIED:

```
  success = load(fname, &if_.eip, &if_.esp);

  /* If load failed, quit. */
  if (!success)
  {
    /*normal execution*/
    palloc_free_page(file_name_);
    thread_exit();
  }
  else
  {
    /*if it was a success instead we need push all needed arguments onto the
        stack
      which we delegate to a helper function*/
    push_on_stack(&if_.esp, file_name_);
    palloc_free_page(file_name_);
  }
```

NOT MODIFIED:

```
  /* Start the user process by simulating a return from an
     interrupt, implemented by intr_exit (in
     threads/intr-stubs.S).  Because intr_exit takes all of its
     arguments on the stack in the form of a `struct intr_frame',
     we just point the stack pointer (%esp) to our stack frame
     and jump to it. */
  asm volatile("movl %0, %%esp; jmp intr_exit" : : "g"(&if_) : "memory");
  NOT_REACHED();
}
```

In this function we need to load a user process and start it, in order to do it we initially load the executable by its name and then we need to pass the parameters on the stack. If the load was successful we will actually push all the needed parameters onto the stack, otherwise if the load fails we will simply wait.

- int process_wait (*modified)*

```
/* Waits for thread TID to die and returns its exit status.  If
   it was terminated by the kernel (i.e. killed due to an
   exception), returns -1.  If TID is invalid or if it was not a
   child of the calling process, or if process_wait() has already
   been successfully called for the given TID, returns -1
   immediately, without waiting.

   This function will be implemented in problem 2-2.  For now, it
   does nothing. */
```

```c
int process_wait(tid_t child_tid)
{
#ifdef USERPROG

  /* disable interrupts for safety*/
  enum intr_level old_level = intr_disable();

  /* we retrieve the child thread thank to a helper function*/
  struct thread *child = thread_get_by_tid(child_tid);
  /* proper check that check if the child is an invalid child for which in
     case a -1 will be returned*/
  if (child == NULL || child->parent != thread_current())
    return -1;

  /* toggle thread field which is the parent wait field stating that the
     parent has called wait on the child*/
  child->pwait = true;
  /* block the parent thread so that it waits child termination */
  thread_block();

  /*re enable interrupts*/
  intr_set_level(old_level);

  /*return the child process status*/
  return child->exit_status;
#else
  /* otherwise just leave implementation as was before*/
  return -1;
#endif
}
```

In this function we have included specific checks for the USERPROG case. We basically obtain the child thread and check if there is a relation with the parent, then if the parent has called wait on the child it is blocked until the child terminates. When the child finishes we return the exit status of the child thread.

- void process_exit (*modified)*

  NOT MODIFIED:

```c

/* Free the current process's resources. */
void
process_exit (void)
{
  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
```

```
19          directory, or our active page directory will be one
20          that's been freed (and cleared). */
21       cur->pagedir = NULL;
22       pagedir_activate (NULL);
23       pagedir_destroy (pd);
24     }
```

MODIFIED:

```
1   /* In order for the test to pass, we need to print the exit status. */
2   printf("%s: exit(%d)\n", cur->name, cur->exit_status);
3
4   /* Check if the parent called wait, if so then we need to unblock him */
5   if (cur->pwait)
6     thread_unblock(cur->parent);
7 }
```

We added the printing of the exit status required for the tests to pass and then we check whether the parent of the process was waiting for it to terminate, if so then we need to unblock the parent, otherwise we just do nothing and the child process will be a zombie process until the parent will call wait on it.