2023

OS Group Project Docs

ARGUMENT PASSING

NAFFAH ABDULLA RASHEED – S1900924

Introduction

The group project of Pintos involves implementing argument passing. At the end of worksheet 3, Pintos can execute based on file name. But the passed arguments are discarded. It is considered argument passing when the arguments extracted, are added to the stack provided by the esp stack pointer. Previously, in worksheet 3, the program name is extracted in the process_execute() function. The function calls are as state below.

process_execute() -> start_process() -> load() -> setup_stack()

The file reading using the program name, is actually handled by the load(). Therefore, I have moved the program name extraction logic into the load function along with argument extraction logic. The setup stack is called; however, we first need to pass the extracted arguments and program name into the function to add them to the stack. The code changed I have brought are described below.

Code Changes

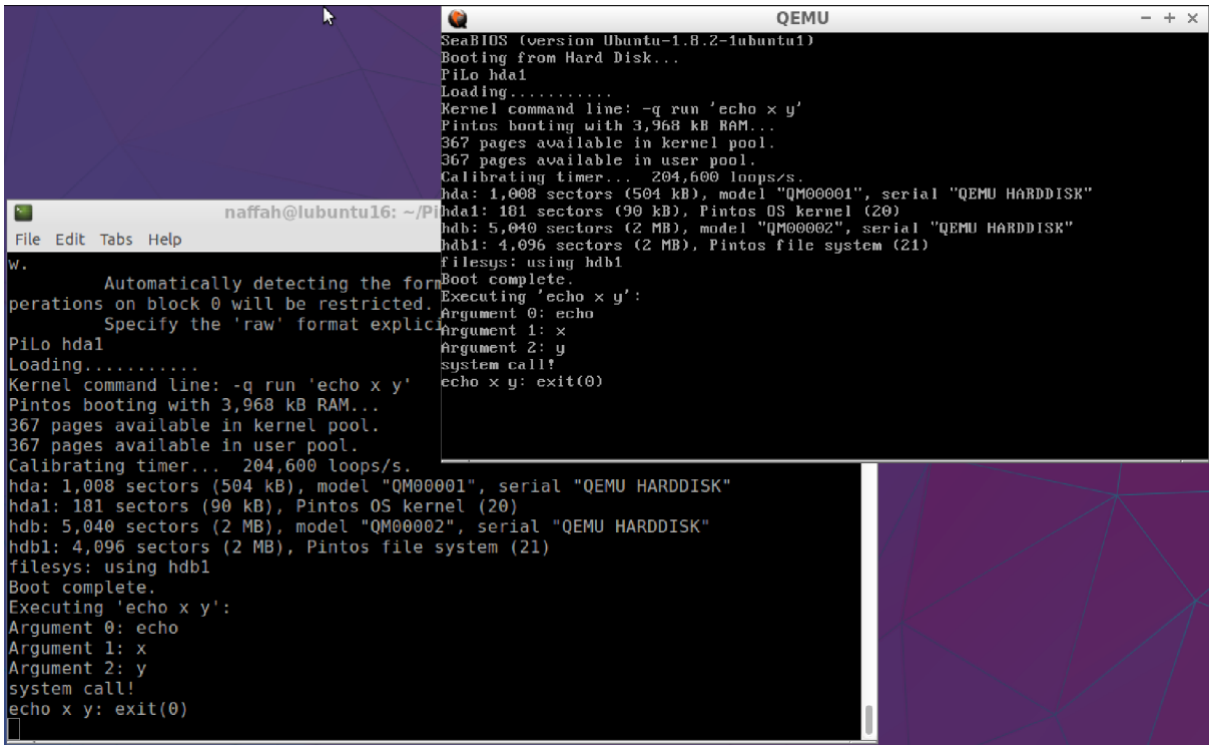
1. The program name is first extracted and added to argv[0]. The remaning pointers in the array is used to store strings of arguments. The argc is simply used to track the number of arguments

```
225   /*
226     Extract arguments from file
227   */
228   char *argv[100]; // Maximum number of arguments
229   int argc = 0;
230   char *save_ptr;
231   argv[0] = strtok_r(file_name, " ", &save_ptr);
232   char *token;
233
234   while ((token = strtok_r(NULL, " ", &save_ptr)) != NULL) {
235     argv[++argc] = token;
236   }
237
238   // for (int i = 0; i <= argc; ++i) {
239   //   printf("Argument %d: %s\n", i, argv[i]);
240   // }
241
```

2. All instances where the file_name was used is replaced with argv[0] containing the extracted program name

```
248   /* Open executable file. */
249   file = filesys_open (argv[0]);
250
251   if (file == NULL)
252     {
253   printf ("load: %s: open failed\n", argv[0]);
254       goto done;
255     }
256
257   /* Read and verify executable header. */
258   if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
259       || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
260       || ehdr.e_type != 2
261       || ehdr.e_machine != 3
262       || ehdr.e_version != 1
263       || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
264       || ehdr.e_phnum > 1024)
265     {
266       printf ("load: %s: error loading executable\n", argv[0]);
267       goto done;
268     }
```

3. To make sure it works, I will simply print the arguments.



4. Next step is to modify setup_stack() function to allow passing of the extracted arguments

```
/* Create a minimal stack by mapping a zeroed page at the top of
   user virtual memory. */
static bool
setup_stack (void **esp, char **argv, int argc) ...
```

5. The setup_stack() is then modified to allow passing arguments to the stack:

    a. The first loop: This loop iterates through the arguments in reverse order (from the last argument to the first). For each argument, it calculates the length of the argument string (including the null terminator), subtracts the space required from `esp` (the stack pointer), stores the resulting `esp` value in `arr[i]`, and then copies the argument string to the calculated address on the stack.
    b. The second loop: This loop iterates through the `arr` array in reverse order (from the last argument to the first). For each argument, it subtracts 4 bytes (the size of a 32-bit pointer) from `esp` to create space for a pointer to the argument's address. Then, it assigns the value of `arr[i]` (which is the address of the argument on the stack) to the location pointed to by `esp`.

```
// Loop through arguments in reverse order to set up the stack
for (int i = argc - 1; i >= 0; --i) {
    // Calculate the space needed for the argument string (including null terminator)
    size_t arg_size = (strlen(argv[i]) + 1) * sizeof(char);

    // Adjust the stack pointer to make space for the argument string
    *esp = *esp - arg_size;

    // Store the current stack pointer in the array for future reference
    arr[i] = (uint32_t *)*esp;

    // Copy the argument string to the stack at the current stack pointer
    memcpy(*esp, argv[i], arg_size);
}

// Loop through the argument pointers on the stack to set up the stack further
for (int i = argc - 1; i >= 0; --i) {
    // Reserve space for a 32-bit pointer on the stack
    *esp = *esp - sizeof(uint32_t);

    // Store the address of the argument in the array at this location
    (*(uint32_t **)(*esp)) = arr[i];
}
```