# Chapter 1

# Design of Module *Userprog*

## 1.1 Assignment Requirements

### 1.1.1 Requirements

The major requirements of the "Userprog" assignment, inspired from the original Pintos documentation, are the following:

- *System Calls for Process Management.* You have to implement the system calls *SyscallProcessExit()*, *SyscallProcessCreate()*, *SyscallProcessGetPid()*, *SyscallProcessWaitForTermination()* and *SyscallProcessCloseHandle()*.
- *System Calls for Thread Management.* You have to implement the system calls *SyscallThreadExit()*, *SyscallThreadCreate()*, *SyscallThreadGetTid()*, *SyscallThreadWaitForTermination()* and *SyscallThreadCloseHandle()*.
- *Program Argument Passing.* You have to change new process creation, such that the program arguments to be passed on its user-space stack.
- *System Calls for File System Access.* You have to implement system calls for opening existing files or creating new files (*SyscallFileCreate()*), reading data from a file *SyscallFileRead()* and closing a file *SyscallFileClose()*).

The way to allocate requirements on member teams.

- 3-members teams
  1. Pálko Loránd Árpád: argument passing + validation of system call arguments (pointers)
  2. Rusu Vlad: system calls for process management + file system access
  3. Szabó Cristian-Iacob: system calls for thread management

## 1.2 Design Description

### 1.2.1 Needed Data Structures and Functions

```
struct _PROCESS { LOCK ChildProcessesLock;
    LIST_ENTRY ChildProcesses;
    LOCK OpenedFileHandlesLock;
    LIST_ENTRY OpenedFileHandles; }
struct _CHILD_PROCESS { PID Pid;
    PID PPid;
    STATUS ExitStatus;
    BOOL Alive;
    LIST_ENTRY ChildProcessesEntry; }
struct _UM_FILE_HANDLE { UM_HANDLE Handle;
    LIST_ENTRY OpenedFileHandlesEntry;
    PFILE_OBJECT File; }
```

### 1.2.2 Detailed Functionality

Some questions you have to answer (inspired from the original Pintos design templates):

1. Argument passing
   - The full command line will be split into an array of strings (*char ***) representing the arguments. On the user stack there will be placed the number of arguments (the length of the arguments array), the pointer to the arguments array, the array of pointers of the actual arguments, and the actual arguments. The arguments will be placed in reversed order.
   - In order to avoid overflowing the stack page we will impose limits on the command line length to PAGE_SIZE / 2 and the number of arguments to 128.
   - *strtok_s()* is reentrant and avoids going over the string's boundary.
2. System calls
   - When a process wants to create/open a file, a handle is associated to the file. The handle can be used only within the process that was assigned the file handle, so the handles are unique within a process. A file can be opened by another process as well and it will be assigned a different handle. Even if a process opened a file, it can open the same file again and being assigned another handle to it.
   - The read/write operations between user and kernel space is done via system calls. The system call performs a memcpy to copy the data. The arguments must be validated at the beginning of the function,

i.e. the pointer must belong to the user space, it must be nonnull, must be mapped, and the calling process must have the rights to perform the requested operation on that memory location.

- There will be exactly 4 inspections of the page table: we need to check the first and the last address of the source page and the destination page. Even if the call wants to copy only 2 bytes the same rule applies. For copying 1 byte there are needed only two checks (source and destination address). These numbers cannot be improved.
- "SyscallProcessWaitForTermination" will check if the process to be waited for exists and is still running. If so, the thread calling the syscall will be blocked until that process terminates. Otherwise the syscall returns. If any resource that is held by the calling thread is requested by a thread in the process which is being waited for, it is considered a programming error and must be avoided by the user. A process can wait for any other process and if the process to be waited for is not in the process hierarchy, it is considered a programming error and must be avoided by the user.
- The easiest way to handle an error in a single place is to mark the error handling section of the function with a label and use a goto instruction when an error occurs. There all the resources will be freed and a proper error code is returned. For example if "SyscallFileRead" is called and the user provides a buffer which is shorter than the read content, the error flag and a specific error code is set, and the error handling section is executed where all the resources are freed and the calling process is terminated. The behavior of the error handling can be customized in function of the error code that was set.