SDP 2022 - OS161 laboratory – 2

(ATTENTION: MP4 videos refer to OS161 version 2.0.2. For version 2.0.3 it is sufficient to replace os161-base-2.0.2 with os161-base-2.0.3. No other changes should be necessary, apart from remembering that the run folder is os161/root instead of pds-os161/root)

In order to face this laboratory you need:

- ***having carried out (and understood) laboratory 1 and, above all, pay close attention to details such as, for example, compilation errors, use of .h files, directives to the precompiler, management of options.***
- ***having seen (and at least understood broadly) the os161-overview and os161-memory lessons***
- ***use the debugger.***

Make a user program executable in OS161

The OS161 user programs, that can be called using the "`p program`" command, for example "`p testbin/palin`", "`p sbin/reboot`", cannot be executed correctly, as in the basic version of OS161, support for the following features is missing:

- `read` and `write` system calls,
- `exit` (`_exit`) system call,
- virtual memory management with free of memory previously allocated to processes,
- `main` arguments (`argc, argv`).

The `read` and `write` system calls, though not strictly necessary to activate a user process, are needed in case the process carries out I/O operations (it happens in almost all programs). The proposed test programs generally perform output, rarely inputs, so the support for `write` is more important.

At the end of a program (end of `main`) the `exit` system call is called (implicitly, when not done explicitly), which causes a system crash, as there is no support for this operation.

We ask to implement the previous points (`main` arguments excluded), in a partial way, such as to allow the execution of a user program with simple I/O and final release of the memory allocated for the process. Since the support for the `main` arguments is NOT required (for this laboratory), some programs might not work properly (*those ones requiring arguments passed to the `main`).*

HINTS:

1) in order to understand which user programs are available, go (in directory `os161/os161-base-2.0.3`) in `userland`: directories `userland/testbin`, `userland/bin`, `userland/sbin`, and sub-directories, contain user test programs (and other stuff). Programs can be read/understood, as well as modified and re-compiled (by means of `bmake` and `bmake install`, that makes them available for execution in `pds-os161/root/testbin` (and other equivalent directories). As you cannot use the debugger within a user program (`mips-harvard-os161-gdb` just debugs the kernel), in case you want to debug your code changes, or even the existing program, you could simply compile (e.g. `testbin/palin/palin.c`) with `gcc` (for the Ubuntu system and the Intel/AMD host processor: e.g. `gcc -g -o palin palin.c`) and run/debug programs outside os161 `gdb` (instead of using `mips-harvard-os161-gdb`).

2) Consider how the `menu_execute()` activates a new thread/process, upon calling `cmd_dispatch()`, to respond to the "`p` ..." command. This activates a new kernel thread through `thread_fork()`, in the `common_prog()` function. The main thread (the parent) exits WITHOUT waiting for the generated thread to finish. The new thread executes the `cmd_progthread()` function (which in turn calls `runprogram()`). The `common_prog()` function also creates a process descriptor, using `proc_create_runprogram()`. In the meantime, the main thread has already returned to the menu, waiting for a new input command.

"write" and "read" system calls

It is recommended to create two functions `sys_write()` and `sys_read()` (with parameters that follow the `read` and `write` prototypes[1]). Functions could be implemented, for instance, in a `kern/syscall/file_syscalls.c` file.

We suggest to limit the implementation to the case of IO from *stdin* and *stdout* (possibly *stderr*), avoiding the explicit management of files (proposed in a subsequent laboratory) and resorting instead (as we are limited to text files) to the `putch()` and `getch()` functions (see for example their use in `kprintf()` and `kgets()`).

"_exit" system call

OS161 (basic version) has a problem with the end of a user process, due to two aspects:

- the process, which ends with `exit()`, not yet completed, enters the `syscall()` function, without support for `SYS__exit`, which causes a "*panic*" call;
- the addition of a "`case SYS__exit:`" in `syscall()` would then require a minimal support for the termination action of a process.

Support for `_exit()` (called from the C function `exit()`, implicitly invoked, when not done explicitly, at the end of the `main`) is obtained by working in a similar way to the system call `read()` and `write()`.
The C `exit()` function (`userland/lib/libc/stdlib/exit.c` file) receives a single integer parameter (error/success code).

Its job is to:

- store this code in the expected field (must be added!) of the thread/process descriptor (`struct thread`, see `kern/include/thread.h` and/or `struct proc` in `kern/include/proc.h`). This code should eventually be read by another thread
- end the thread (and the process) that calls `exit()`, calling the `thread_exit()` function (file `thread.c`), which cleans the data structure of the calling thread and sends it into a *zombie* state. The thread descriptor is not destroyed (it is therefore still readable by another thread). The actual

[1]The protothypes of read and write system calls, as well as other library functions that activate system calls (e.g exit), in userland/include/unistd.h (or other .h files). As an alternative, you can use the browsable code or the Linux help (man read or on oppure l'help linux (man command or online documentation): help and online documentation are preferred/suggested as they include an explanation of the behaviour, of the parameters and of the returned value.

destruction will be done by the `thread_destroy`(), called from the `exorcise()` to the next `thread_switch()`.

At the moment, you are not asked to save the state, therefore it is recommended to implement a reduced version of `exit()`. This function must be able at least to call `as_destroy()` and to close the thread: for this purpose it is suggested to create a `sys__exit(int status)` function (for example by creating a `kern/syscall/proc_syscalls.c` file to contain it). The function then will call `as_destroy()` and `thread_exit()`.

The main reason for this partial (and temporary) implementation is to be able to activate and close a user process without destructive errors, making it possible to test the allocation and (above all) the de-allocation of the user memory. In order to "complete" the exit system call, you need a support for synchronization (syscall waitpid), whereas for read and write you need a more adequate support in terms of file system: both topics will be covered in subsequent labs.

Virtual memory management

OS161 (basic version) has a very basic virtual memory manager that only performs contiguous allocation of real memory, without ever releasing it (see the `kern/arch/mips/vm/dumbvm.c` file).
Each time `getppages()/ram_stealmem()` are called, a new interval in physical RAM is allocated, and it will never be released.

We ask you to create a contiguous memory allocator which (or modify the existing one so that), instead of using the current implementations of `getppages()` / `ram_stealmem()`, changes them, by keeping track of allocated/used and free RAM pages (or frames).

We recommend, as a first version, to create a "*bitmap*" or more simply an array of (int or char) flags. This requires changing the search for free RAM performed by `as_prepare_load()` and `alloc_kpages()` (so that a data structure able to record allocated memory is properly setup) and the return of memory by `as_destroy()` and `free_kpages()`. We suggest modifying the `getppages()` and/or the `ram_stealmem()` functions (for contiguous physical memory allocation). WARNING: `getppages()` is called both by `kmalloc() -> alloc_kpages()` (for kernel dynamic allocation) and by `as_prepare_load()` to allocate memory for user processes.

PROPOSED SOLUTION

We provide a few files (`dumbvm.c`, `syscall.c`, `file_syscalls.c`, `proc_syscalls.c`, `syscall.h`), that partially accomplish the proposed work. The files require the definition of the "*syscalls*" option in `conf.kern`, which makes the `proc_syscalls.c` and `file_syscalls.c` files optional.

The proposed implementation is partial, so improvements and/or alternatives are possible.
WARNING: for any use, it is necessary to place the provided files in the correct folders (completion of the first lab is thus very important, as well as a correct understanding of how to manage options and .h files!) and to define the syscalls option in `conf.kern` (as previously explained).

To test / debug the developed program, we suggest to execute (with debug and breakpoints in the functions of interest) functions that allocate and de-allocate memory, for example thread tests (*t1*, *tt2*, *tt3*), for `kmalloc()/kfree()`, or user programs to generate and free the address space of a process.

We suggest some additional tasks related to the memory allocator ***(WARNING: the tasks listed below are optional, not so simple/easy to solve; so we leave the student the choice to do them, based on his/her ability/skills):***

1. Full support of `kfree()`: the proposed version does not release the memory allocated by `kfree()` before `vm_bootstrap()` activates the allocation tables. Study a solution that could solve this problem.

2. Create variants of the proposed scheme, for example obtain all the RAM available at the bootstrap, then always manage allocation and deallocation using the tables.

3. Use of a real bitmap, instead of a char array, for the `freeRamFrames` table. A bitmap as a bit array is usually created as an `unsigned int` or `unsigned long int` array, and requires the use of appropriate operators to access the individual bits (shift and masks with bitwise or/and operators). A simple explanation of how to make bit arrays in C can be found here: http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html .

4. Create a function to print statistics on allocated and free memory. The function is called via a new command (*memstats*) added to the OS161 menu.

5. Improve the search time for a free interval and/or lighten the mutual exclusion requirements. The proposed version has a linear research cost, in which activities are accomplished in mutual exclusion. Furthermore, access to the table is carried out in mutual exclusion. Consider avoiding mutual exclusion in the `isTableActive()` function. Consider reducing the mutual exclusion requirements for accessing the tables.