OS161 - System and Device Programming

LAB02 - User program

- **Author**: `Davide Arcolini`
- **Date**: `2022-03-21`

Table of contents

Getting started

> **NB**: all over the file, `os161-base_2.0.3` is referred as `src`. This is to simplified paths as `os161/os161-base_2.0.3/...` to `os1617src`. Remind it!

The *os161* user programs can be called using the `p program` command (e.g., running `p testbin/palin` or `p sbin/reboot`). However, these processes **can not** be executed correctly due to the lack of implementation of the majority of system calls. As it can be seen in the `syscall.c` file (*path*: `os161/src/kern/arch/mips/syscall/`), there is provided only the implementation of two system calls (`sys_reboot()` and `sys__time()`):

```
void syscall(struct trapframe *tf) {

    /* ... */

    retval = 0;
    switch (callno) {
        case SYS_reboot:
            err = sys_reboot(tf->tf_a0);
        break;

        case SYS___time:
            err = sys___time((userptr_t)tf->tf_a0, (userptr_t)tf->tf_a1);
        break;

        /* HERE GOES THE MISSING IMPLEMENTATIONS */

        default:
            kprintf("Unknown syscall %d\n", callno);
            err = ENOSYS;
        break;
    }
    /* ... */

}
```

The previously shown system calls dispatcher is called by processes which do expect some more implementation. In particular:

1. `sys_read()` and `sys_write()`;
2. `_exit()`;
3. Virtual memory management system calls.

IO system calls

> NB: All the functions and files used in this laboratory have a post-fix `_LAB02` to differentiate them from the "*real*" system function.

In particular, the prototypes of Input/Output system calls, for the scope of this laboratory: `sys_read_LAB02()` and `sys_write_LAB02()` are defined in `syscalls_LAB02.h` (*path*: `os161/src/kern/include/`), as:

```
/**
 * @file syscalls_LAB02.h
 * @author Davide Arcolini (davide.arcolini@studenti.polito.it)
 * @brief Contains the declarations of the system calls used in LAB02 (and
following)
 * @version 0.1
 * @date 2022-05-02
 *
 * @copyright Copyright (c) 2022
 *
*/


#ifndef _SYSCALLS_LAB02_H_
#define _SYSCALLS_LAB02_H_

#include "opt-lab02.h"

#define STDIN_FILENO 0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2

typedef int ssize_t;

/**
 * @brief sys_write_LAB02() function attempts to write n_byte bytes from
the buffer pointed to
 *         by buf to the file associated with the open file descriptor, fd.
If n_byte is 0,
 *         sys_write_LAB02() will return 0 and have no other results if the
```

```
file is a regular file;
 *        otherwise, the results are unspecified.
 *
 * @param fd File descriptor: where to write
 * @param buf pointer to the buffer from which the sys_write_LAB02 will
get the content to write
 * @param n_bytes number of bytes to be written
 * @return ssize_t number of bytes written
*/
#if OPT_LAB02
ssize_t sys_write_LAB02(int fd, const void *buf, ssize_t n_bytes);
#endif

/**
 * @brief sys_read_LAB02() attempts to read up to n_bytes from file
descriptor fd
          into the buffer starting at buf.
 *
 * @param fd File descriptor: read from fd
 * @param buf pointer to the buffer to which the sys_read_LAB02 will store
the content read
 * @param n_bytes number of bytes to be read
 * @return ssize_t number of bytes read
*/
#if OPT_LAB02
ssize_t sys_read_LAB02(int fd, void *buf, ssize_t n_bytes);
#endif

/* ... (here you will find the declaration of sys_exit_LAB02() ... */

#endif /* _SYSCALLS_LAB02_H_ */
```

The file reports the classical structure of a `.h` file:

1. Definition of the dependencies file name (`_SYSCALLS_LAB02_H_`) only in case it has non already be defined somewhere else. This avoid the compiler to execute multiple inclusion of the same header file.
2. Optional include of the compiler-generated library: `opt-lab02.h`.
3. The definition of the prototypes only in case the option `OPT_LAB02` has been activated in the configuration file `conf.kern` (*path*: `os161/src/kern/conf/`).

The actual implementation of the functions is defined in `syscalls_LAB02.c` (*path*: `os161/src/kern/syscalls/`):

```

/**
 * @file syscalls_LAB02.c
 * @author Davide Arcolini (davide.arcolini@studenti.polito.it)
 * @brief Contains the definitions of the system calls used in LAB02 (and
following)
 * @version 0.1
 * @date 2022-05-02
 *
 * @copyright Copyright (c) 2022
 *
*/

#include <types.h>
#include <kern/unistd.h>
#include <clock.h>
#include <copyinout.h>
#include <lib.h>
#include <proc.h>
#include <thread.h>
#include <addrspace.h>

#include "syscalls_LAB02.h"

/**
 * @brief sys_write_LAB02() function attempts to write n_byte bytes from
the buffer pointed to
 *        by buf to the file associated with the open file descriptor, fd.
If n_byte is 0,
 *        sys_write_LAB02() will return 0 and have no other results if the
file is a regular file;
 *        otherwise, the results are unspecified.
 *
 * @param fd File descriptor: where to write
 * @param buf pointer to the buffer from which the sys_write_LAB02 will
get the content to write
 * @param n_bytes number of bytes to be written
 * @return ssize_t number of bytes written
 */
#if OPT_LAB02
ssize_t sys_write_LAB02(int fd, const void *buf, ssize_t n_bytes) {

    /*
        CHECK IMPLEMENTATION CONSTRAINT
        -------------------------------
        NB: sys_write_LAB02 is defined to work only with stdout (and,
eventually, stderr).
            If fd is not stdout or stderr, it will raise an error and
exit.
    */
    if (fd != STDOUT_FILENO && fd != STDERR_FILENO) {
        kprintf("[!] Ayo, I am quite dumb... I can work only with STDOUT
and STDERR...\n");
        return -1;
    }

    /* PRINTING BYTES TO THE FILE DESCRIPTOR FROM THE BUFFER*/
    char *pointer = (char *) buf;
    for (int i = 0; i < n_bytes; i++) {
        putch(pointer[i]);
    }
    /* RETURNING NUMBER OF BYTES WRITTEN */
    return n_bytes;

}
#endif


/**
 * @brief sys_read_LAB02() attempts to read up to n_bytes from file
descriptor fd
          into the buffer starting at buf.
 *
 * @param fd File descriptor: read from fd
 * @param buf pointer to the buffer to which the sys_read_LAB02 will store
the content read
 * @param n_bytes number of bytes to be read
 * @return ssize_t number of bytes read
*/
#if OPT_LAB02
ssize_t sys_read_LAB02(int fd, void *buf, ssize_t n_bytes) {

    /*
        CHECK IMPLEMENTATION CONSTRAINT
        -------------------------------
        NB: sys_write_LAB02 is defined to work only with stdout (and,
eventually, stderr).
            If fd is not stdout or stderr, it will raise an error and
exit.
    */
    if (fd != STDIN_FILENO) {
        kprintf("[!] Ayo, I am quite dumb... I can work only with
STDIN...\n");
        return -1;
    }

    /* READING BYTES FROM THE FILE DESCRIPTOR TO THE BUFFER*/
    char *pointer = (char *) buf;
    for (int i = 0; i < n_bytes; i++) {
        if ((pointer[i] = getch()) == -1) {
            return i;
        }
    }

    /* RETURNING THE NUMBER OF BYTES READ */
    return n_bytes;
}
#endif

/* ... (here you will find the definition of sys_exit_LAB02() ... */
```

The dispatcher is in charge of calling these functions based on the system call number which is passed to the dispatcher (the definitions of the values can be found in `os161/src/kern/include/kern/`):

```
#define SYS_fork        0
#define SYS_vfork       1
#define SYS_execv       2
#define SYS__exit       3
#define SYS_waitpid     4
#define SYS_getpid      5
#define SYS_getppid     6
/* ETC ... */
#define SYS_read        50
/* ETC ... */
#define SYS_write       55
/* ETC ... */
```

The following piece of code is added to the dispatcher in order to provide the correct functionalities:

```
#if OPT_LAB02
        case SYS_write:
            retval = sys_write_LAB02(tf->tf_a0, (const void *) tf->tf_a1,
(ssize_t) tf->tf_a2);
            err = 0;
            if(retval == -1) err = ENOSYS;

        break;

        case SYS_read:
            retval = sys_read_LAB02(tf->tf_a0, (void *) tf->tf_a1,
(ssize_t) tf->tf_a2);
            err = 0;
            if (retval == -1) err = ENOSYS;
        break;

        case SYS__exit:
            sys_exit_LAB02(tf->tf_a0);
            err = 0;
        break;
#endif
```

How IO works

Here it is described, step by step, the whole execution of the user program `testbin/palin`, starting from the moment when user actually write in the command line `p testbin/palin` to the execution of the write system call.

1. Upon starting of the operation system, the `main()` function calls (after the call to `boot()`), the function `menu()`. This function iterates forever (until user indicates with the command `q` the end of execution), asking the user to input some information. i.e. the program to be inserted:

```c
void menu(char *args) {

    char buf[64];
    menu_execute(args, 1);

    while (1) {
        kprintf("OS/161 kernel [? for menu]: ");
        kgets(buf, sizeof(buf));
        menu_execute(buf, 0);
    }
}
```

2. `menu_execute()` receives the input string passed by the user and parse it in order to provide to the dispatcher a formatted data.

```c
static void menu_execute(char *line, int isargs) {

    /* ... */

    result = cmd_dispatch(command);

    /* ... */
}
```

3. `cmd_dispatcher()` receives the formatted directive and (linearly) search inside a table (`cmdtable[]`) the entry which corresponds to the directive passed in by the user. Each entry table has two field:

    - `cmdtable[].name` which stores the command that has to be matched by the input user (in case of `p testbin/palin`, it stores `p` in `cmdtable[6].name`).
    - `cmdtable[].func` which stores the address of the function which has to be called for that particular `cmdtable[].name` (in case of `p testbin/palin`, it stores `0x8000bfa8 <cmd_prog>` in `cmdtable[6].func`).

    Once the right entry is found, the dispatcher calls the `cmdtable[].func` with the necessary argument (received formatted by the `menu_execute()`).

    ```c
    result = cmdtable[i].func(nargs, args);
    ```

4. `cmd_prog()` removed the leading `p` from the command passed and calls `common_prog()`.

```c
return common_prog(nargs, args);
```

5. `common_prog()` is in charge of creating a new process to run the program of the user. It starts from a **kernel thread** (which will become a user process soon) to which the `cmd_progthread` routine is passed. Note that this does not wait for the subprogram to finish, but returns immediately to the menu. This is usually not what you want, so you should have it call your system-calls-assignment `waitpid` code after forking.

```
static int common_prog(int nargs, char **args) {
    struct proc *proc;
    int result;

    /* Create a process for the new program to run in. */
    proc = proc_create_runprogram(args[0] /* name */);
    if (proc == NULL) {
        return ENOMEM;
    }

    result = thread_fork(args[0] /* thread name */,
            proc /* new process */,
            cmd_progthread /* thread function */,
            args /* thread arg */, nargs /* thread arg */);
    if (result) {
        kprintf("thread_fork failed: %s\n", strerror(result));
        proc_destroy(proc);
        return result;
    }

    /*
     * The new process will be destroyed when the program exits...
     * once you write the code for handling that.
     */

    return 0;
}
```

6. `cmd_progthread()` prepares the process information and it is in charge of calling `runprogram()` with the program name once everything is ready.

```
result = runprogram(progname);
```

> **HERE STARTS THE REAL EXECUTION OF THE USER PROCESS**

7. `runprogram()` is in charge of running the new process. In particular, it performs the following operations:

   - Open the file (i.e. `progname`).

     ```
     result = vfs_open(progname, O_RDONLY, 0, &v);
     ```

- Create a new address space.

  ```
  as = as_create();
  ```

- Switch to the new address space and activate it.

  ```
  proc_setas(as);
  as_activate();
  ```

- Load the executable at the entry-point in order to let the kernel know where the user process starts.

  ```
  result = load_elf(v, &entrypoint);
  ```

- Define the user stack in the address space.

  ```
  result = as_define_stack(as, &stackptr);
  ```

- Warp to user mode. Here the actual execution starts. **Note**: there is no coming back from here. This is why a `panic()` message can be found after the following command:

  ```
  enter_new_process(0         /*argc*/,
                    NULL      /*userspace addr of argv*/,
                    NULL      /*userspace addr of environment*/,
                    stackptr,
                    entrypoint);

  /* enter_new_process does not return. */
  panic("enter_new_process returned\n");
  return EINVAL;
  ```

  The user mode is set by a function inside `enter_new_process()` called `mips_usermode()`.

  ```
  void enter_new_process(int argc, userptr_t argv, userptr_t env,
  vaddr_t stack, vaddr_t entry) {
      struct trapframe tf;

      bzero(&tf, sizeof(tf));

      tf.tf_status = CST_IRQMASK | CST_IEp | CST_KUp;
  ```

```
        tf.tf_epc = entry;
        tf.tf_a0 = argc;
        tf.tf_a1 = (vaddr_t)argv;
        tf.tf_a2 = (vaddr_t)env;
        tf.tf_sp = stack;

        mips_usermode(&tf);
    }
```

8. The program `testbin/palin` starts now the execution. Actually, who cares about what this program does... What really is important here is that `testbin/palin` happens to call `printf()`, which is nothing but a *fancy* wrapper of the `write()` function. This function itself is a wrapper to the **system call** `SYS_write` which is triggered by calling the `syscall(struct trapframe *tf)` function in `syscall.c` (*path*: `os161/src/kern/arch/mips/syscall/`).
   In particular, the `write()` function creates a `trapframe *tf` which contains the arguments passed to the write function, along with the dispatcher value:

   ```
   callno = tf->tf_v0;
   ```

   which stores the number `55` corresponding to the `SYS_write`. Now, we are able to program `syscall.c` to call our `sys_write_LAB02()` when the `printf()` function is called from up above.

EXIT System Call

> NB: All the functions and files used in this laboratory have a post-fix `_LAB02` to differentiate them from the "*real*" system function.

In particular, the prototypes of EXIT system calls, for the scope of this laboratory: `sys_exit_LAB02()`is defined in `syscalls_LAB02.h` (*path*: `os161/src/kern/include/`), as:

```
/**
 * @brief Basic (simpler) implementation of _exit(). It should saves the state of the process
 *        ending (in order to be read from other processes), release memory and destroy the
 *        main thread. However, this simpler version only destroy the main thread and then exit.
 *
 * @param status integer value indicating the result of the operation. Tipically EXIT_SUCCESS or
 *               EXIT_FAILURE
 */
#if OPT_LAB02
void sys_exit_LAB02(int status);
#endif
```

The actual implementation of the functions is defined in `syscalls_LAB02.c` (*path*: `os161/src/kern/syscalls/`):

```
/**
 * @brief Basic (simpler) implementation of _exit(). It should saves the
state of the process
 *        ending (in order to be read from other processes), release
memory and destroy the
 *        main thread. However, this simpler version only destroy the main
thread and then exit.
 *
 * @param status integer value indicating the result of the operation.
Tipically EXIT_SUCCESS or
 *               EXIT_FAILURE
 */
#if OPT_LAB02
void sys_exit_LAB02(int status) {

    /* GET THE ADDRESS SPACE OF THE CURRENT PROCESS */
    struct addrspace *as = proc_getas();
    if (as == NULL) {
        kprintf("[!] proc_getas() returned NULL value.\n");
        return;
    }

    /* DESTORY THE ADDRESS SPACE */
    as_destroy(as);

    /* HERE YOU SHOULD SAVE SOME STATE BEFORE EXITING, BUT FOR NOW, WHO
CARES... */

    /* MAIN THREAD TERMINATES HERE. BYE BYE */
    thread_exit();

    /* WAIT! YOU SHOULD NOT HAPPEN TO BE HERE */
    panic("[!] Wait! You should not be here. Some errors happened during
thread_exit()...\n");

    /* JUST TO HANDLE WARNINGS ABOUT UNUSED STATUS */
    (void) status;

}
#endif
```

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. As a matter of fact, inside the `kmain()` function, there is a call to the `boot()` function which performs the bootstrap of the whole system. In particular, it performs some **early initialization** (in which memory for the kernel is allocated):

```
ram_bootstrap();
proc_bootstrap();
thread_bootstrap();
hardclock_bootstrap();
vfs_bootstrap();
kheap_nextgeneration();
```

and some **late initialization**, among which there is a call to `vm_bootstrap()`:

```
vm_bootstrap();
kprintf_bootstrap();
thread_start_cpus();
```

Initially, this function was completely empty, because no virtual memory management system was implemented. In order to implement the management of allocation and release of the memory, it is necessary to adopt a different strategy rather than the simple `ram_stealmem()` used.

1. **Define the bitmap**. The **bitmap** is a pair of arrays composed as follows:

- **`unsigned int *bitmap_freePages;`**: an `unsigned int *` variable storing a boolean value `0`/`1`, indicating if the i-page of memory is ready to be re-used or it is still being referred (`{0: 'not_free', 1: 'free'}`).

> **NB (from Cabodi's lecture)**: here `not_free` has a double meaning: it can refers to an area of memory which **has never been allocated before** (e.g., the portion of memory storing the kernel, for which the `bitmap_freePages` has been set to zero but never reset to `1`, since the kernel never release it's memory area during execution) or also to an area of memory which is **currently used by some process**.
> However, this is only one implementation possible. Another solution would have been to let the array `bitmap_freePages` starts mapping the memory after the kernel, so that no space in storing `0`s would have ever being wasted.
> Also, the choice of `1` and `0` is up to you.

- **`unsigned long *bitmap_sizePages;`**: an `unsigned long *` variable storing the allocation size required for the process granted the i-page.

```
static unsigned int *bitmap_freePages = NULL;        /* BITMAP STORING THE
POSITION OF THE FREE PAGES */
static unsigned long *bitmap_sizePages = NULL;       /* BITMAP STORING THE
SIZE REQUIRED FOR THE ALLOCATION */
```

The bitmap allows to record where the memory has been used and for how much space.

2. **`vm_bootstrap()`: initialization of data** `vm_bootstrap()` is in charge of initialize all the necessary data structure used by the bitmap and the consequent management system. In particular, it

performs:

- Computation of the number of pages in the available memory (i.e., after the kernel has reserved its space) through a call of the `ram_getsize()` function.
- Allocation of the bitmap.
- Initialization of the bitmap: the memory in the bitmap is initialize as `not_free`. It will be freed by the release mechanism of the processes which follows the call to this function.
- Enabling of the boolean variable `allocTableActive`, which indicates whether the `vm_bootstrap` has already been performed or not. This is due to the fact that upon calling of this function (and consequently before the activation of the boolean variable), the other functions like `getppages()` should beware like no virtual memory management system is present.

> **NB**: the boolean flag is accessed in **mutual exclusion**, to avoid the need of more sophisticated synchronization mechanism. One of the *optional* task of this laboratory is to change this behavior.

```c
/**
 * @brief Bootstrap the virtual memory and manage allocation
 *
 */
void vm_bootstrap(void) {

    /* DISCOVER HOW MANY PAGES CAN BE AVAILABLE IN RAM */
    number_ram_pages = ((int) ram_getsize()) / PAGE_SIZE;

    /* ALLOCATING BITMAP */
    bitmap_freePages = (unsigned int *) kmalloc(number_ram_pages *
sizeof(unsigned int));
    bitmap_sizePages = (unsigned long *) kmalloc(number_ram_pages *
sizeof(unsigned long));
    if (bitmap_freePages == NULL || bitmap_sizePages == NULL) {
        kprintf("[ERROR] kmalloc() failed execution");
        bitmap_freePages = NULL;
        bitmap_sizePages = NULL;
        return;
    }

    /* INITIALIZE BITMAP */
    for (int i = 0; i < number_ram_pages; i++) {
        bitmap_freePages[i] = 0;                    /* {0: 'not_free'} */
        bitmap_sizePages[i] = 0;                    /* {size: 0} */
    }

    /* ENABLING ALLOCATION TABLE */
    spinlock_acquire(&freemem_lock);
    allocTableActive = 1;
    spinlock_release(&freemem_lock);

    return;
}
```

3. **Requesting of new memory** When a process is created, new space in memory is reserved for the execution of that process. In particular, the function `proc_create()` is called, inside which we can find the function `kmalloc()` which is in charge of allocating space in memory for the process. This last function calls itself the `getppages()` function. `getppages()` retrieves the address in which space is memory starts, based on the amount of pages (`unsigned long npages`) required by the process. In particular:
   - If it is able to retrieve free pages from the bitmap, it simply updates the bitmap structure for the following requests;
   - If it is *not* able to retrieve free pages, it performs the old `ram_stealmem()` call.

```
static paddr_t getppages(unsigned long npages) {

    /* GETTING PHYSICAL ADDRESS OF THE FREE PAGES */
    paddr_t addr = getfreeppages(npages);

    if (addr == 0) {

        /* NO FREEE PAGES FOUND */
        spinlock_acquire(&stealmem_lock);
        addr = ram_stealmem(npages);
        spinlock_release(&stealmem_lock);
    }

    if (addr != 0 && isTableActive()) {

        /* UPDATING BITMAP */
        spinlock_acquire(&freemem_lock);
        bitmap_sizePages[addr/PAGE_SIZE] = npages;              /*
ALLOCATION OF THE VIRTUAL SPACE */
        spinlock_release(&freemem_lock);
    }

    return addr;
}
```

4. **The new `getfreeppages()` function** The `getppages()` function performs as a first operation, an attempt to get free pages from the bitmap. This is done through the new defined function `getfreepages()`. In particular:
   - It searches through the `bitmap_freePages` if there are actually pages with a `{1: 'free'}` flag.
   - It search if, starting from a `free` position, there is enough space to satisfy the process memory-space requirements.

     > **NB**: this search is performed with a linear complexity. One of the *optional* task of this laboratory is to improve this complexity.

   - It updates the bitmap consequently.
   - It returns the address found (`0` in case no free pages were found).

```
/**
 * @brief Return the physical address of the memory based on a bitmap-
allocation strategy
 *        Moreover, it updates the underlaying data structure.
 *
 * @param npages number of pages required to be allocated
 * @return paddr_t physical address
 */
static paddr_t getfreeppages(unsigned long npages) {

    paddr_t addr = 0;

    /* CHECKING IF ALLOCATION TABLE IS ACTIVE (i.e., can use the LAB02
methods) */
    if (!isTableActive()) {
        return 0;
    }

    /* GETTING LOCK ON MEMORY */
    spinlock_acquire(&freemem_lock);

    /* ALGORITHM FOR SEARCHING FREE PAGES */
    long int start_tmp = -1;
    long int start_adr = -1;
    for (long int i = 0; i < number_ram_pages; i++) {

        /* CHECKING IF THE POSITION IS AVAILABLE */
        if (bitmap_freePages[i]) {

            /* CASE IN WHICH START IS THE BEGINNING OF THE BITMAP */
            if (i == 0 || bitmap_freePages[i-1] == 0) {
                start_tmp = i;
            }

            /* CHECKING IF THE AVAILABLE POSITION IS SUFFICIENT FROM THE
LAST START FOUND */
            if (i - start_tmp + 1 >= (long int)npages) {
                start_adr = start_tmp;
                break;
            }
        }
    }

    /* CHECKING IF WE HAVE FOUND AN AVAILABLE SPACE */
    if (start_adr != -1) {
        for (long int i = 0; i < start_adr + (long int)npages; i++) {
            bitmap_freePages[i] = 0;                 /* UPDATING BITMAP OF
POSITIONS */
        }
        bitmap_sizePages[start_adr] = npages;        /* UPDATING BITMAP OF
DIMENSIONS */
        addr = (paddr_t) start_adr * PAGE_SIZE;
    } else {
        addr = 0;
    }

    /* RELEASING SPINLOCK */
    spinlock_release(&freemem_lock);

    return addr;
}
```

5. **Release of memory** On the other hand, when a process ends its execution, it calls the `free_kpages()` function. Previously, this function was empty due to the fact that *os161* performs no release of memory. Now, this function checks if the `vm_bootstrap()` has already been performed and, in case, it performs two operations:
   - Computes the physical address of the memory

     ```
     paddr_t paddr = addr - MIPS_KSEG0;
     long int start_addr = paddr/PAGE_SIZE;
     ```

   - Actually set the pages as `free` in the bitmap.

```
void free_kpages(vaddr_t addr) {

    /* SEE IF FREE IS ABILITATED */
    if (isTableActive()) {

        /* COMPUTING PHYSICAL ADDRESS */
        paddr_t paddr = addr - MIPS_KSEG0;
        long int start_addr = paddr/PAGE_SIZE;

        /* FREEING THE PAGES */
        freeppages(paddr, bitmap_sizePages[start_addr]);

    } else {
        (void) addr;
    }
}
```

6. **The new `freeppages()` function** The `freeppages()` function performs a the set to `1` of the `bitmap_freePages` data structure, in order to indicates that those pages are free to be used from now on:

```
static void freeppages(paddr_t paddr, unsigned long size) {

    /* CHECKING IF TABLE IS ACTIVE */
    if (!isTableActive()) {
```

```
            return;
        }

        /* COMPUTING PHYSICAL ADDRESS */
        unsigned long start_addr = paddr/PAGE_SIZE;

        /* GET LOCK ON MEMORY */
        spinlock_acquire(&freemem_lock);

        /* FREEING (VIRTUALLY OBV) THE PAGES */
        for (unsigned long i = start_addr; i < start_addr + size; i++) {
            bitmap_freePages[i] = 1;
        }

        /* RELEASE LOCK */
        spinlock_release(&freemem_lock);

        return;
    }
```

Conclusions

The bitmap improves the memory management of the *os161* operating system. It is important to understand that this is a simplified solution. The *optional* tasks provide support for more complicated solutions.