Group 2 Document

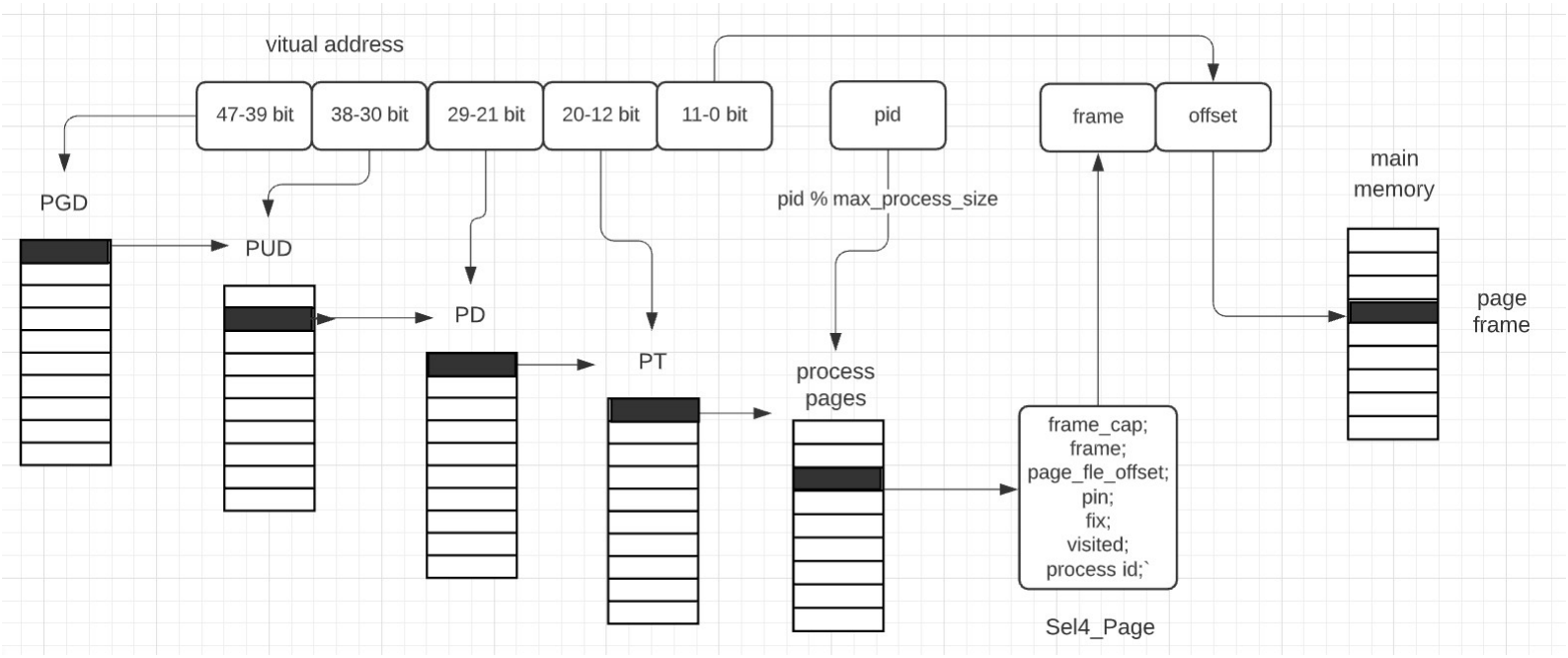
ChengRui Wang z5142652, Yuzhang Wang z5272202

1 Execution mode: events

Syscall loop (in main.c) waits for events and calls an appropriate event handler based on the system call number.Our system contains 14 system calls.They are SOS_WRITE , SOS_READ, SOS_SLEEP, SOS_TIME_STAMP, SOS_OPEN, SOS_CLOSE, SOS_BRK, SOS_STAT, SOS_GET_DIRENT, SOS_MY_PID, SOS_PROCESS_CREAT, SOS_PROCESS_STATUS, SO_PROCESS_DETELE and SOS_PROCESS_WAIT. Each of them has an event handler. A handler is a function that runs until completion and returns to the event loop.
All the handlers are defined in "systask.c" where we make the system call a task. The task object is a data structure that contains a reply object and other information. When it is finished, the reply object will be used to reply to the client using "seL4_send()". Syscalls are not blocked by the server. Because the function "handle_syscall" in "main.c" returns a bool value. If it is blocked, it returns true. Then a new reply should be generated for the other system call. This is done in the function "syscall_loop" in "main.c"

2 Pagetable structure:

Defined in "app_mapping.h".
The page table is a organnized in four-levels, where all level are 4K in size and use 9 bits of the virtual address. All levels contain 512 entries to the next level page table. The paging structures are as follows:
Page Global Directory (PGD): the top-level paging structure (the root of the virtual address space). It contains 512 pointers to the PUD.
Page Upper Directory (PUD): the second level paging structure. It contains the corresponding untype, the frame cap, the frame, and 512 pointers to the PD.
Page Directory (PD): the third level paging structure. It contains the corresponding untype, the frame cap, the frame, and 512 pointers to the PT.
Page Table (PT): the fourth level paging structure. It contains the corresponding untype, the frame cap, and the frame. It also contains 16 pointers to the Sel4_Page. It contains the corresponding pages of all the processes. (maximum 16 processes).
The picture below shows the 4 level page table and the process pages array.

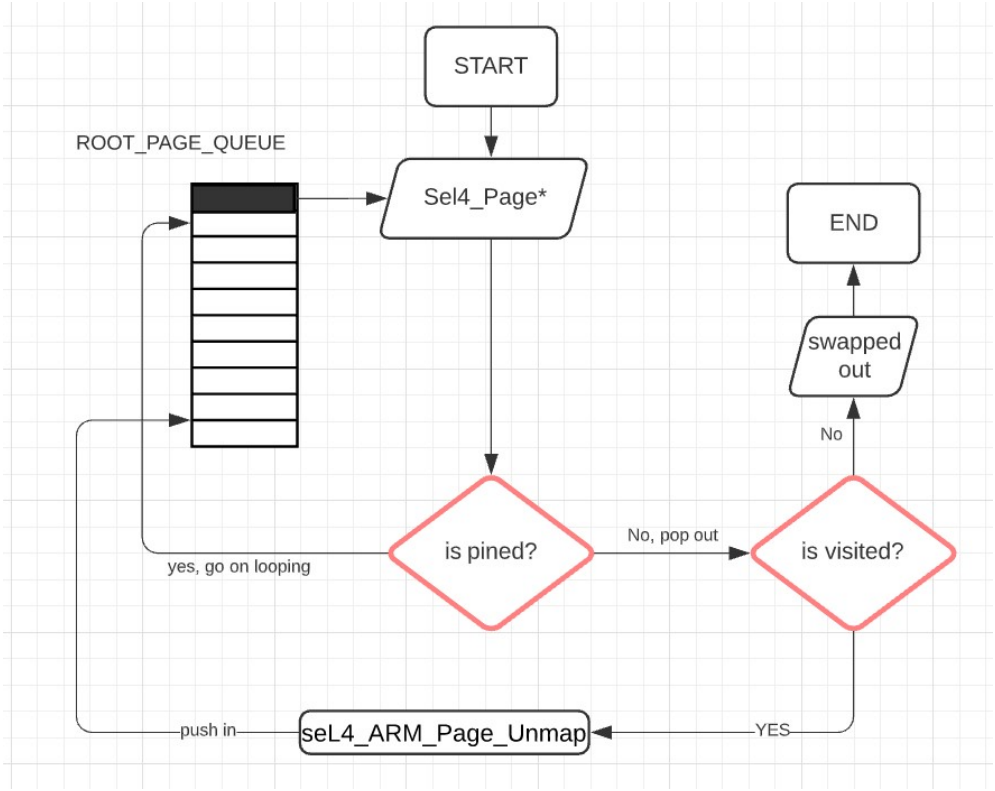


The page structure is called " Sel4_Page". Each page contains the following attributes: the frame and the frame _cap, the virtual address, a size_t value "page_file_offset"(the offset in the page file, a bool value "pin" (indicated if it can be swapped out), a bool value "visited" (used for the second chance popping strategy) and a sel4_Word "pid"(process id).

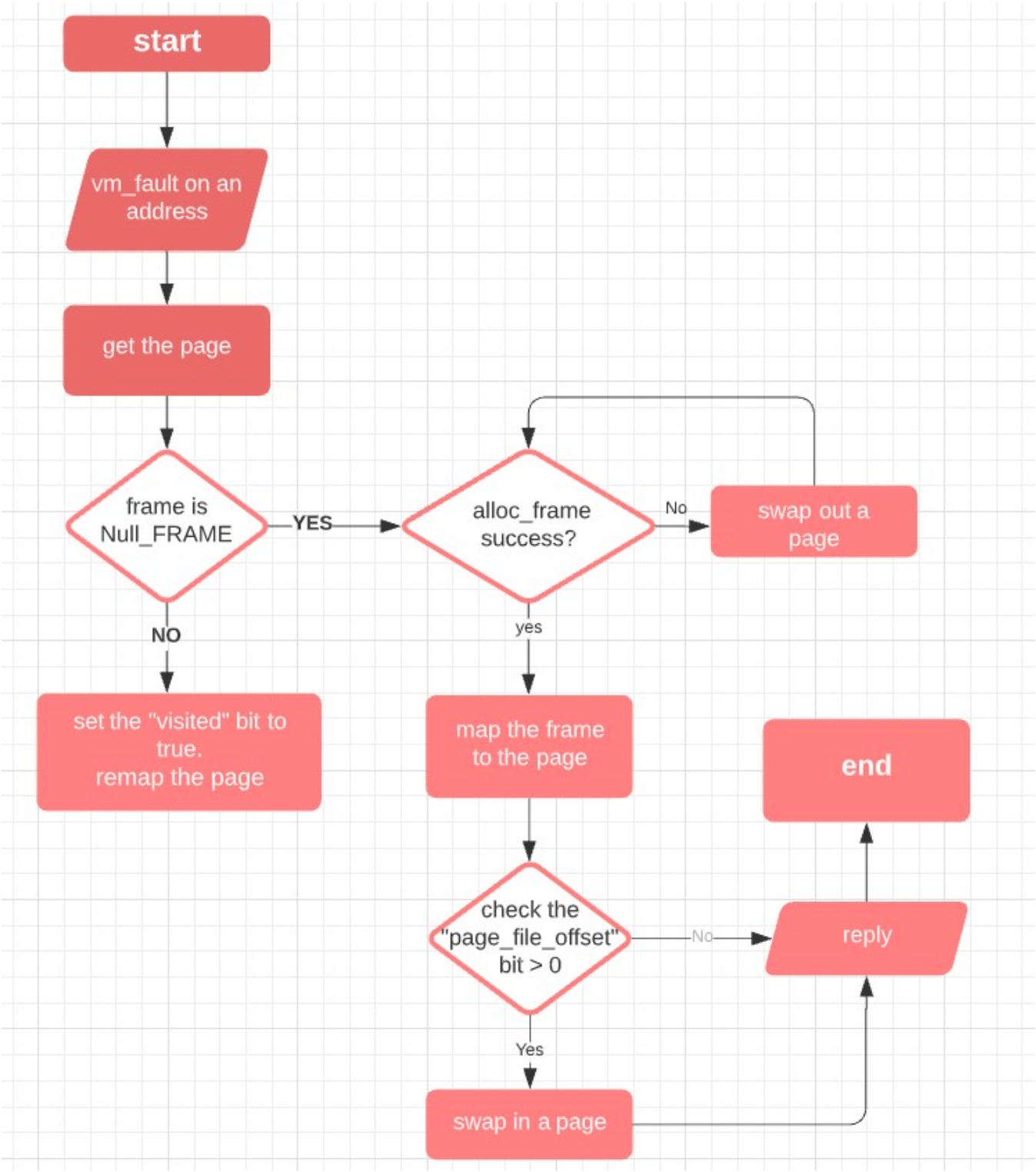
3 Demand paging

Defined in "paging.h".
The "SwapOutTask" and "SwapInTask" are two data structures for swapping in and swapping. They contain the page that needs to be swapped in/ swapped out, a callback function, and other information.
We use an array named "page_file_list" to keep track of the page slots in the NFS file "pagefile". This can help us in two ways:
1 . First when a page needs to be swapped out, we can find which slot is empty and put the page there.
2. Second when a page in the NFS file needs to be swapped in, the corresponding page can be found because each page saves the offset in the NFS file.

3.1 Swap out a page:



To implement the second chance popping, we define a queue named "ROOT_PAGE_QUEUE" to save current pages in the memory. When a page needs to be swapped out, the system loops through every page in the queue. Each page has two bits for the popping, "visited" and "pin". The "visited" indicates the page was visited recently. And the "pin" bit indicates whether or not this page can be swapped out.
The function "swap_out_one page" first checks the "pin" bit is set to false:
if not, just skip it. Else check if its "visited" bit is set. If the "visited" bit is True, it means this page should not be swapped out. Then use " seL4_ARM_Page_Unmap" to unmap it, set its "visited" to false, and go on looping. These pages are still in the memory. It will be remapped when page fault happens (shown in the picture below), then we use "seL4_ARM_Page_map " to remap it, and also set its "visited" bit to True, which means it is visited recently.
If the "visited" bit is false, then this page should be swapped out. So basically we find the first page whose visited bit is false and swap it out.
On the other hand, some pages are pined in the queue and can not be swapped out. They are pages for the kernel object like IPC buffer and kernel stack.

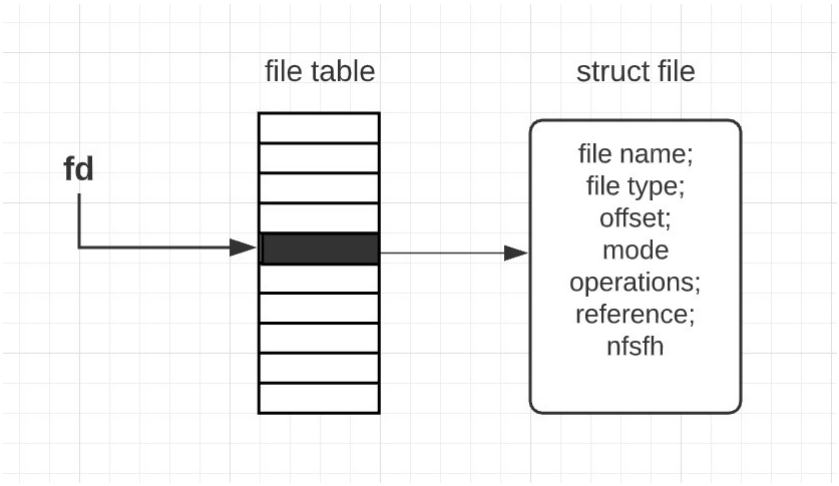

start
vm_fault on an address
get the page
frame is Null_FRAME
YES
NO
alloc_frame success?
No
swap out a page
yes
set the "visited" bit to true.
remap the page
map the frame to the page
end
check the "page_file_offset" bit > 0
No
reply
Yes
swap in a page

3.2 Swap in a page:

When a page is visited, we first check its "page_file_offset" bit. This bit indicates its position in the page file.
If it is 0, then do nothing.
If it is greater than 0, it means it needs to be swapped in. Then read the page from the page file in the corresponding offset.
The above picture shows this.

4 I/O subsystem

Defined in "nfs_api.c" and "console.c".
We create a "struct file" array that behaves like a global open file array. Each process has its own file descriptor table. The index of the array is the file descriptor and the "struct file" is like a vnode. This structure is defined in "fdtable.h". The "struct file" contains the file pointer and the other information.

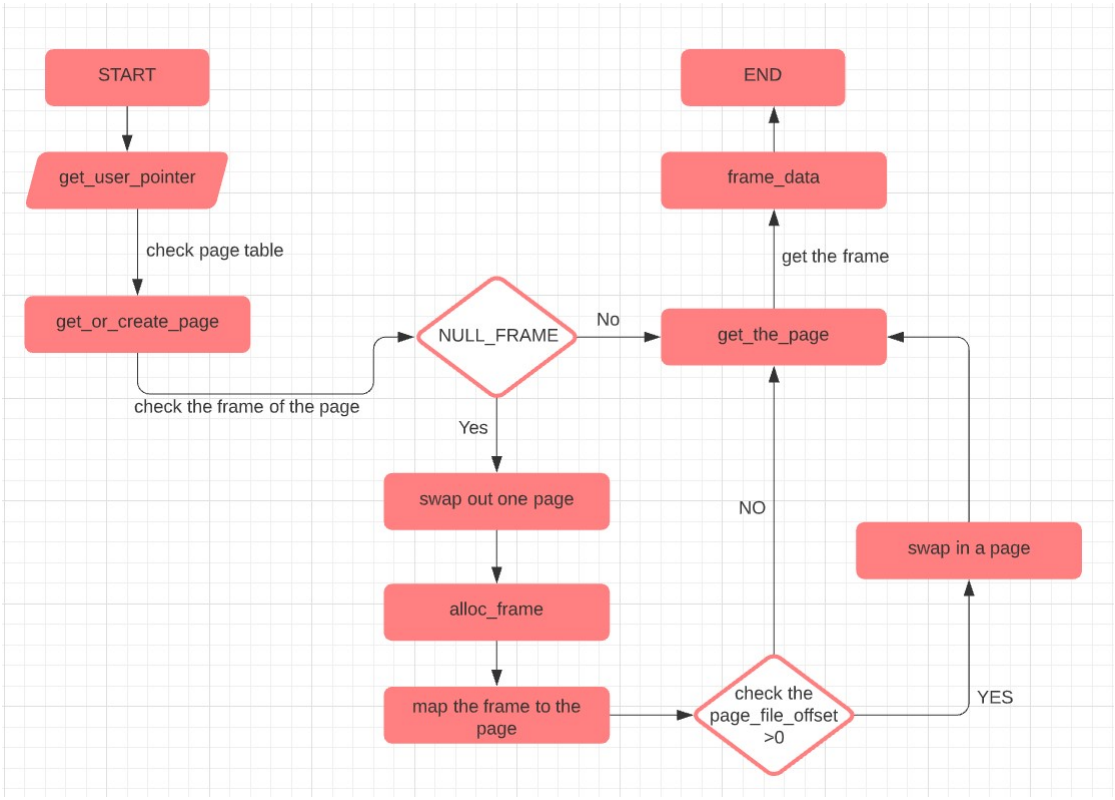


Each file contains a "struct file_opts_t" which stores some operation function pointers.

```
typedef struct file_opts {
    int (*read) (ReadTask *task);
    int (*write) (WriteTask *task);
    int (*open) (const char * name, int mode);
    int (*close) (int fd);
} file_opts_t;
```

ReadTask and WriteTask are two data structures used for reading and writing. Instead of using an IPC buffer to transfer data, we use the method " sos maps a user page into itself." Like the

picture below.



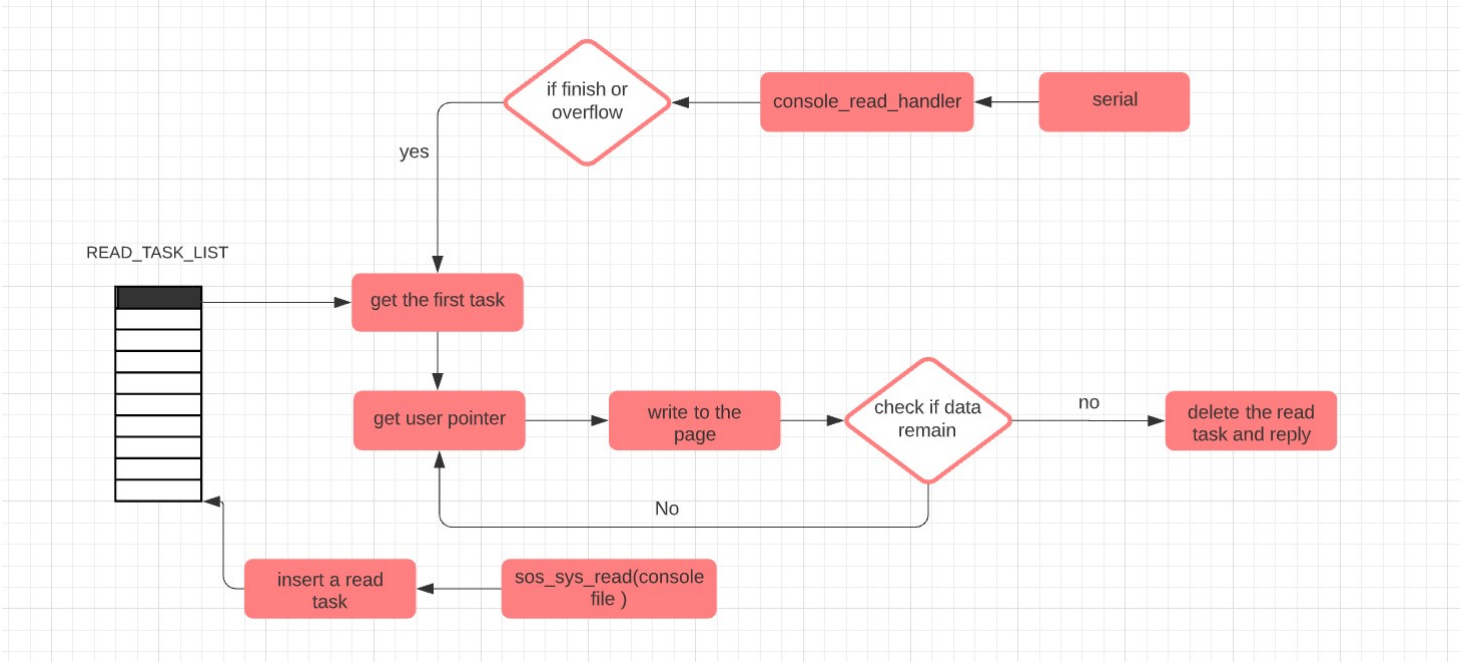
To achieve this method, we define a function named "get_or_create_page" in "appmapping.h". It first lookups at the page table to which page the user pointer belongs. Then it gets the contents of a frame and stores them in the Readtask and Writetask, These two data structures will be passed into the reading and writing functions as arguments. Finally, when reading data from files or writing data to files, these pages can be used directly.

4.1 Open:

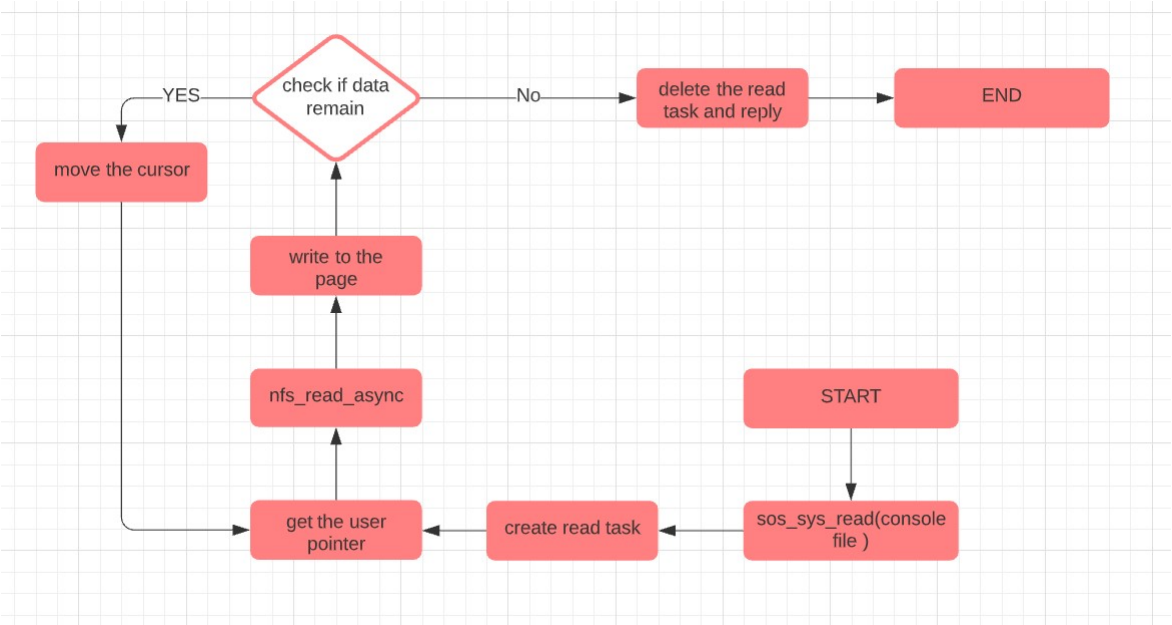
We open the console file at boot, in "main.c". When a "sos_sys_open" is called, we can check if the file is opened by finding the corresponding file descriptor. If found, just reply to the client. If not, then the function "nfs_open_file " , defined in "nfs_api.c", will do the open task. (Because the console file has already opened, so the file descriptor must be found).

4.2 Read

If the "sos_sys_read" is called on the "console", a ReadTask is inserted into an array named ReadTaskList. The callback function in the "serial_register_handler" will first find the unfinished Readtask in the ReadTaskList and reply to it. If a ReadTask is finished, it will be removed from the ReadTaskList. Like the picture below.



If the system reads an "NFS" file, it will use the read function named "nfs_read_api" defined in the nfs_api.c. The callback function in the "nfs_pread_aysnc " will modify the file pointer.Like the picture below.



4.3 Write

"sos_sys_write" is similar to the "sos_sys_read" except that console is a multiple writer so WriteTask doesn't need to be stored.

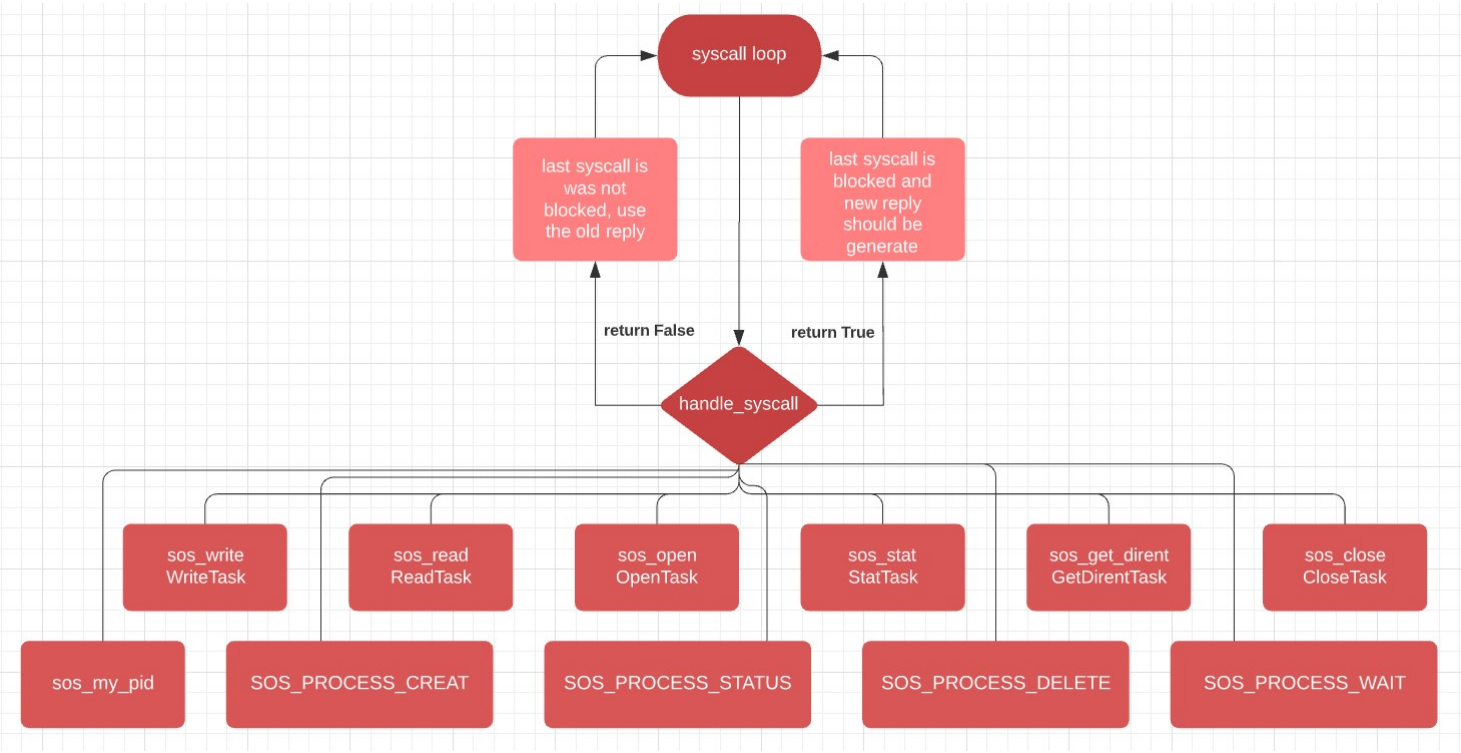
4.4 Close:

Close the file by the file descriptor. The file descriptor is the index to the file array, free it and decrease the file reference by 1.

4.5 get_dirent and stat:

We use nfs_opendir_async and nfs_stat64_async. We define two data structures for them in read_task.h, GetDirentTask and StatTask.

5 System call dispatching

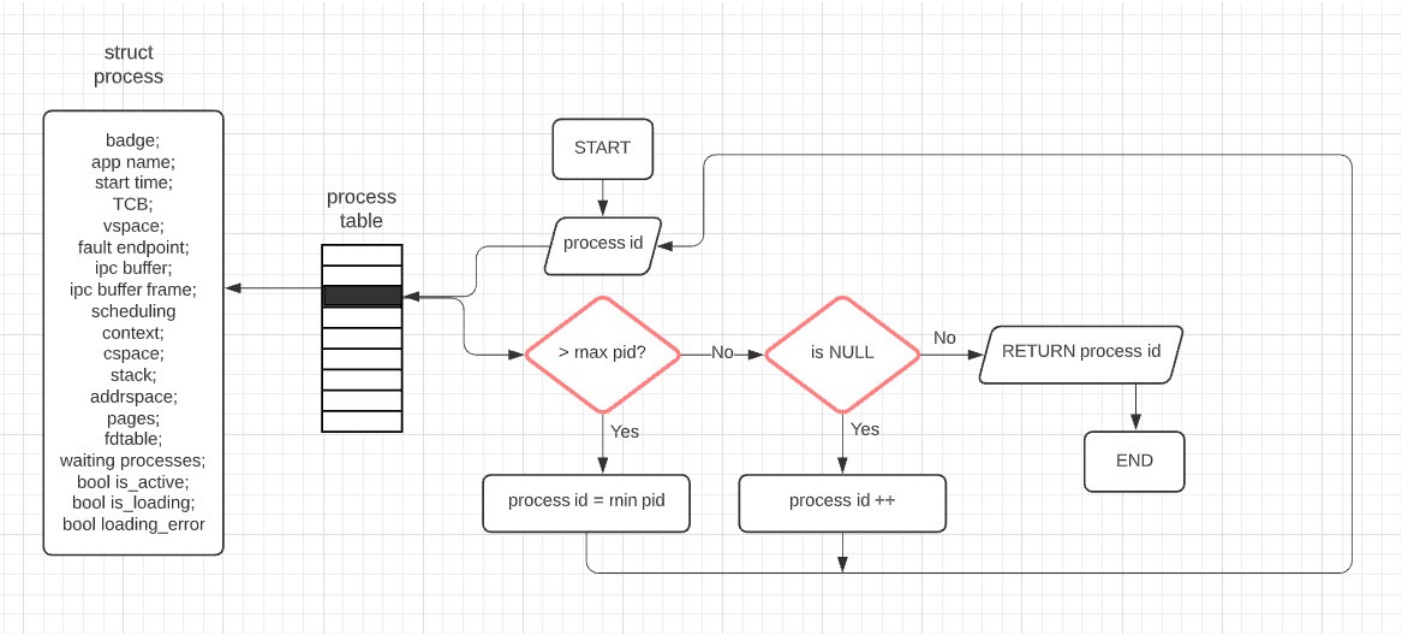


The handle_syscall in "main.c" returns a bool value. As the above diagram shows, some system calls will be blocked, then a new reply should be generated for the other system call.
The dispatching file is "systask.c".
Task objects store the reply object. So that the corresponding system call can reply to the client

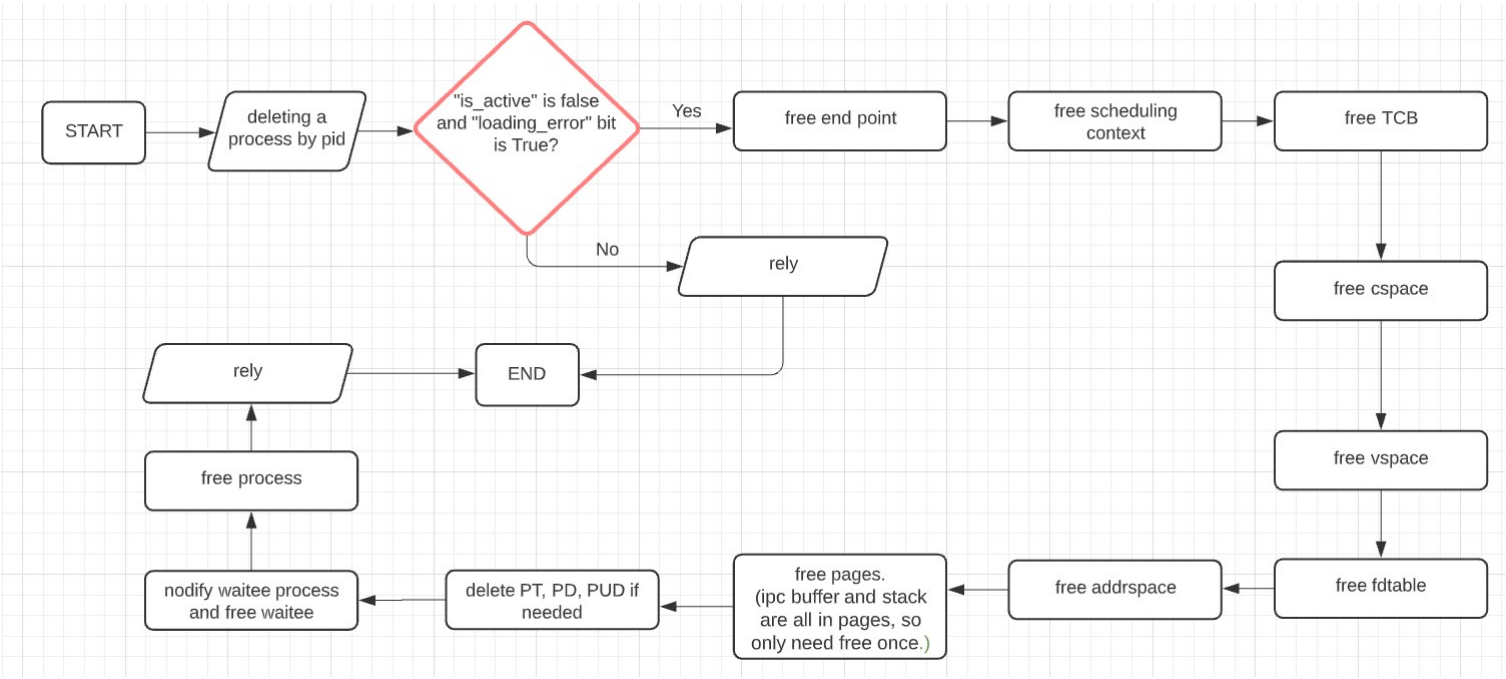
when finished. These Task objects are all defined in read_task.h.

6 Process management

We use an array to store all processes. The pid is the index of the array.
The following picture shows the structure of the process and how we get a new process id.

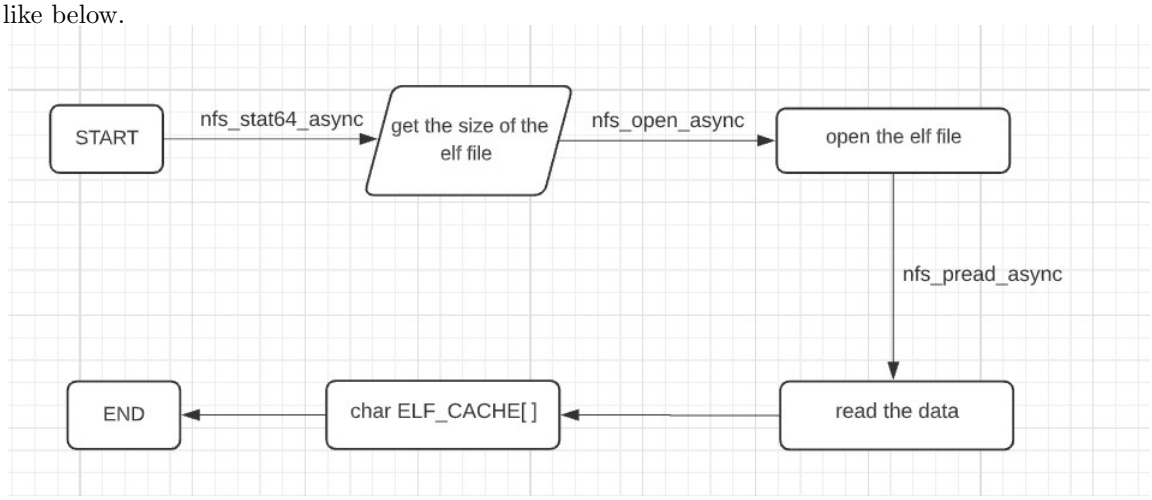


When deleting a process, we need to clean up the process first, and then notify the waiting processes.

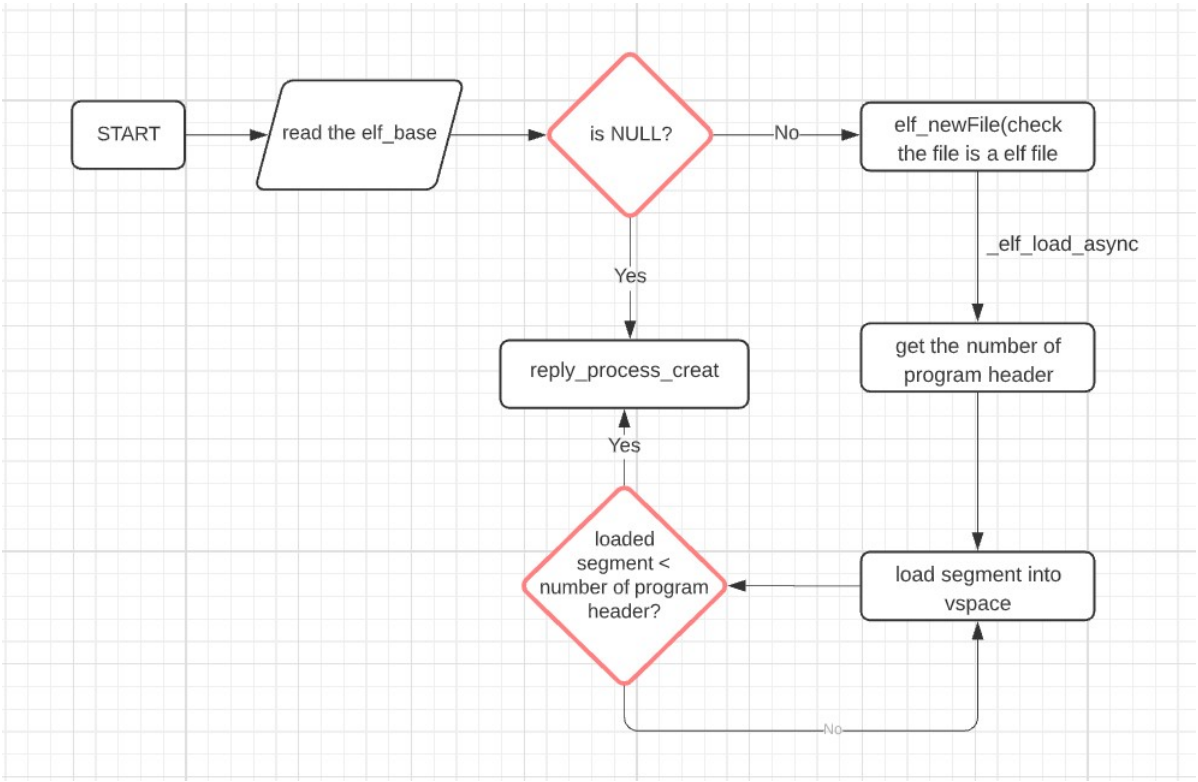


Load_elf from NFS:

We read the elf file from NFS once, and then write it to frames page by page. The "ELF_CACHE" defined in systask.h is a char array, used to read in the elf file data from the NFS. The reading is like below.



And then load the elf file:


START
read the elf_base
is NULL?
No
elf_newFile(check the file is a elf file
_elf_load_async
Yes
reply_process_creat
get the number of program header
Yes
loaded segment < number of program header?
load segment into vspace
No