Roadmap

Using the Roadmap

This roadmap is divided into stages. Each stage is to be done in sequential order. You will build eXpOS incrementally. Links are provided throughout the document for further reference. There are two kinds of links. The contents of the **important links** must be read immediately before proceeding with the roadmap. The informative links may be clicked for more information about a particular concept. However this information may not be necessary at that point and you may proceed with the roadmap without visiting these links.

It is very important that you proceed with the roadmap on a regular schedule and not get lost in the links. Hence, an approximate amount of time (in hours) which you are expected to spend on each stage is noted along with the stage. If you find that reading a particular documentation/link takes too much time, skip it for the time being and come back to it only when needed.

Preparatory Stages:

The preparatory stages help you to get familiarized with the disk bootstrap loading process, disk access mechanism, file-system specification, debugger, paging hardware, interrupt handling mechanism, program loading, library linkage and function calling conventions, application binary interface (ABI), context switching between applications and so forth. You will need 2-3 weeks to complete these stages.

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

Stage 1 : Setting up the System (https://exposnitc.github.io/Roadmap.html#collapse1)

Stage 2 : Understanding the Filesystem (2 Hours) (https://exposnitc.github.io/Roadmap.html#collapse2)

Stage 3 : Bootstrap Loader (2 Hours) (https://exposnitc.github.io/Roadmap.html#collapse3)

Stage 4 : Learning the SPL Language (2 Hours) (https://exposnitc.github.io/Roadmap.html#collapse4)

Stage 5 : XSM Debugging (2 Hours) (https://exposnitc.github.io/Roadmap.html#collapse5)

Stage 6 : Running a user program (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse6)

Stage 7 : ABI and XEXE Format (2 Hours) (https://exposnitc.github.io/Roadmap.html#collapse7)

(https://exposnitc.github.io/Roadmap.html#collapse7)

(https://exposnitc.github.io/Roadmap.html#collapse7)

Stage 8 : Handling Timer Interrupt (2 Hours) (https://exposnitc.github.io/Roadmap.html#collapse8)

Stage 9 : Handling kernel stack (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse9)

Stage 10 : Console output (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse10)

Stage 11 : Introduction to ExpL (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse11)

Stage 12 : Introduction to Multiprogramming (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse12)

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

Intermediate Stages:

In these stages, you will come across more advanced hardware features like, disk interrupt handling and exception handling. You will be implementing some basic kernel subsystems that will be used throughout the project. You will modularize your kernel into functional subsystems for resource management, memory management, device management, etc. You will implement a primitive user interface (Shell) and the final version of the OS loader (Exec system call). The amount of implementation details given in the road map will gradually diminish and many details will be left to be worked out by you. You wil need 3-4 weeks to complete these stages.

Stage 13 : Boot Module (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse13)

Stage 14 : Round robin scheduler (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse14)

Stage 15 : Resource Manager Module (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse15)

Stage 16 : Console Input (6 Hours) (https://exposnitc.github.io/Roadmap.html#collapse16)

Stage 17 : Program Loader (6 Hours) (https://exposnitc.github.io/Roadmap.html#collapse17)

Stage 18 : Disk Interrupt Handler (6 Hours) (https://exposnitc.github.io/Roadmap.html#collapse18)

Stage 19 : Exception Handler (6 Hours) (https://exposnitc.github.io/Roadmap.html#collapse19)

Final Stages:

Stage 20-Stage 27 are the final stages of the project where you will implement all the system calls stipulated in the ABI documentation. Typically 5-6 weeks will be needed to complete these stages. At the end of the twentieth stage, basic system calls for process creation and termination – Fork, Exec and Exit will be completed. The next

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

two stages take up system calls implementing signals and semaphores. The next three stages address the implementation of the file system. The subsequent stages add multi-user support and virtual memory support. (An advanced stage (Stage 28) describing how the OS can be ported to a two-core extension of the XSM machine has been added subsequently.)

Stage 20 : Process Creation and Termination (12 Hours) (https://exposnitc.github.io/Roadmap.html#collapse20)

Stage 21 : Process Synchronization (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse21)

Stage 22 : Semaphores (4 Hours) (https://exposnitc.github.io/Roadmap.html#collapse22)

Stage 23 : File Creation and Deletion (6 Hours) (https://exposnitc.github.io/Roadmap.html#collapse23)

Stage 24 : File Read (12 Hours) (https://exposnitc.github.io/Roadmap.html#collapse24)

Stage 25 : File Write (12 Hours) (https://exposnitc.github.io/Roadmap.html#collapse25)

Stage 26 : User Management (12 Hours) (https://exposnitc.github.io/Roadmap.html#collapse26)

Stage 27 : Pager Module (18 Hours) (https://exposnitc.github.io/Roadmap.html#collapse27)

Learning Objectives (https://exposnitc.github.io/Roadmap.html#lo27)

Understand the disk swap-out and swap-in mechanisms.

Implement the pager module that supports Swap in and Swap out functions.

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

Pre-requisite Reading (https://exposnitc.github.io/Roadmap.html#lo27a)

Revisit the description of data structures- Process table (https://exposnitc.github.io/os_design-files/process_table.html), Page table (https://exposnitc.github.io/os_design-files/process_table.html#per_page_table), System status table (https://exposnitc.github.io/os_design-files/mem_ds.html#ss_table), Disk Map table (https://exposnitc.github.io/os_design-files/process_table.html#disk_map_table).

In this stage, we will learn how the limited physical memory pages of the XSM machine can be used effectively to run the maximum number of concurrent processes. To achieve this, we will implement the functions **Swap Out** and **Swap In** of Pager module (https://exposnitc.github.io/os_modules/Module_6.html) (Module 6). Corresponding modifications are done in Timer Interrupt (https://exposnitc.github.io/os_design-files/timer.html) and Context Switch module (https://exposnitc.github.io/os_modules/Module_5.html) as well.

eXpOS gives provision to execute 16 processes concurrently in the system and the number of memory pages available for user processes are 52 (from 76 to 127 - See memory organization (https://exposnitc.github.io/os_implementation.html)). Consider a case, where every process uses four code, two heap, two user stack and one kernel stack pages. Then each process will need 9 memory pages. In this situation, the OS will run out of memory before all 16 processes can be brought into the memory, as the memory required will be 144 pages in total. A solution to this problem is following - when the OS falls short of free memory pages needed for a process, try to identify some inactive process whose memory pages could be swapped out to the disk. The memory pages freed this way can be allocated to the new process. At a later point of time, the OS can swap back the swapped out pages when the inactive process needs to be re-activated. This gives illusion of more memory than actual available memory. (Also see Virtual Memory (https://en.wikipedia.org/wiki/Virtual_memory).)

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

eXpOS uses an approach for memory management where the system does not wait for all the memory to become completely exhausted before initiating a process swap out. Instead, the OS regularly checks for the status of available memory. **At any time if the OS finds that the available (free) memory drops below a critical level, a**

swap out is initiated. In such case, the OS identifies a relatively inactive process and swaps out some of the pages of the process to make more free memory available. The critical level in eXpOS is denoted by MEM_LOW (https://exposnitc.github.io/support_tools-files/constants.html) (MEM_LOW is equal to 4 in present design). When available memory pages are less than MEM_LOW, eXpOS calls **Swap Out** function of pager module (https://exposnitc.github.io/os_modules/Module_6.html). *Swap Out* function selects a suitable process to swap out to the disk. The memory pages used by the selected process are moved into the disk blocks and the memory pages (except the memory pages of the library) are released. The code pages are not required to be copied to the disk as the disk already contains a copy of the code pages. The kernel stack page of a process is also not swapped out by eXpOS. However, the heap and the user stack pages are swapped out into the disk. eXpOS has 256 reserved blocks in the disk (256 to 511 - see disk organization (https://exposnitc.github.io/os_implementation.html)) for swapping purpose. This area is called **swap area**.

A swapped out process is swapped back into memory, when one of the following events occur:

1) A process has remained in swapped out state for more than a threshold amount of time.

2) The available memory pages exceed certain level denoted by MEM_HIGH (https://exposnitc.github.io/support_tools-files/constants.html) (MEM_HIGH is set to 12 in present design).

Each process has an associated TICK value (see process table (https://exposnitc.github.io/os_design-files/process_table.html)) which is reset whenever the process is swapped out. The TICK value is incremented every time the system enters the timer interrupt routine. If the TICK value of a swapped out process exceeds the value MAX_TICK (https://exposnitc.github.io/support_tools-files/constants.html#swap), the OS decides that the process must be swapped in. A second condition when the OS decides that a process can be swapped in is when the available number of free memory pages (see MEM_FREE_COUNT in system status table (https://exposnitc.github.io/os_design-files/mem_ds.html#ss_table)) exceeds the value MEM_HIGH.

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

When does the OS check for MEM_LOW/MEM_HIGH condition? This is done in the timer interrupt handler (https://exposnitc.github.io/os_design-files/timer.html). Since the system enters the timer routine at regular intervals, this design ensures that regular monitoring of TICK/MEM_FREE_COUNT is achieved.

We will modify the timer interrupt handler in the following way. Whenever it is entered from the context of any process **except** a special **swapper daemon process** (to be described later), the handler will inspect the TICK status of the swapped out processes and the memory availability status in the system status table to decide whether a swap-in/swap-out must be initiated. If swap-in/swap-out is needed, the timer will set the PAGING_STATUS field in the system status table to SWAP_IN/SWAP_OUT appropriately to inform the scheduler about the need for a swap-in/swap-out. The timer handler then passes control to the scheduler. Note that the timer does not initiate any swap-in/swap-out now. We will describe the actions performed when the timer interrupt handler is entered from the context of the swapper daemon soon below.

We will modify the eXpOS scheduler to schedule the swapper daemon whenever PAGING_STATUS field in the system status table is set to SWAP_IN/SWAP_OUT. The OS reserves PID=15 for the swapper daemon process. (Thus the swapper daemon joins login, shell and idle processess as special processess initiated by the kernel.) **The swapper daemon shares the code of the idle process, and is essentially a duplicate idle process running with a different PID. Its sole purpose is to set up a user context for swapping operations.** A consequence of the introduction of the swapper daemon is that only 12 user applications can run concurrently now.

If the timer interrupt handler is entered from the context of the swapper daemon, then it will call the Swap-in/Swap-out functions of the pager module after inspecting the value of PAGING_STATUS in the system status table. **Thus, swap-in/swap out will be initiated by the timer interrupt handler only from the context of the swapper daemon.**

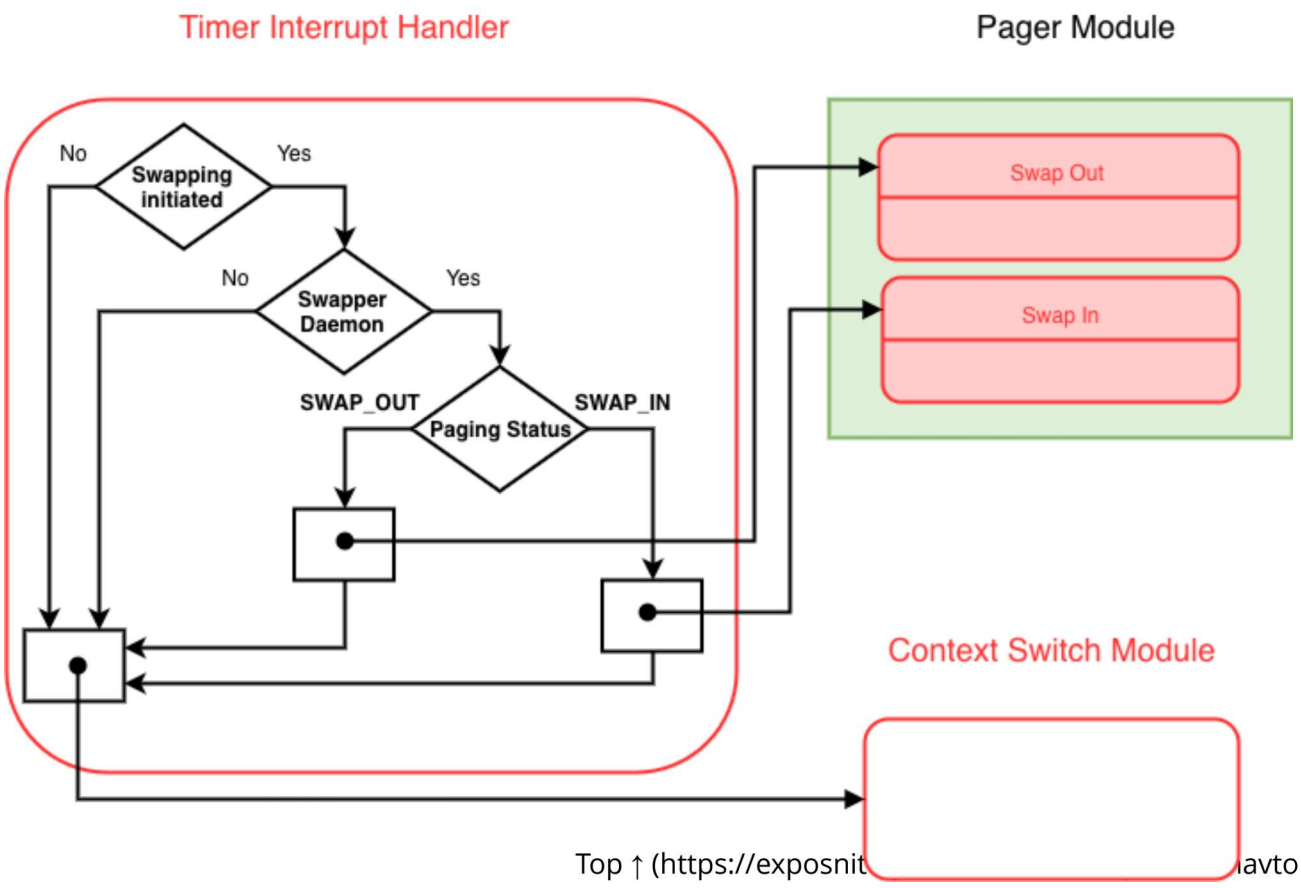
While swapping is ongoing, the swapper daemon may get blocked when swap-in/swap-out operation waits for a disk-memory transfer. The OS scheduler will run the Idle process in such case. Note that the Idle process will never get blocked, and can always be scheduled whenever no other process is ready to run.

Top | (https://exposnitc.github.io/Roadmap.html#navtop)

Once a swap-in/swap-out is initiated from the timer, the OS scheduler will not schedule any process other than the swapper daemon or the idle process until the swap-in/swap-out is completed. This policy is taken to avoid unpredicatable conditions that can arise if other processes rapidly acquire/release memory and change the memory availability in the system while a swap operation is ongoing. This design, though not very efficient, is simple to implement and yet achieves the goal of having the full quota of 16 process in concurrent execution. (Note that the size of the process table in the eXpOS implementation outlined here limits the number of concurrent processes to 16).

The algorithms for Swap-in and Swap-out are implemented in the Pager Module (https://exposnitc.github.io/os_modules/Module_6.html) (Module 6).

Modifications to Timer Interrupt



Top ↑ (https://exposnit avtop)

Control flow for timer interrupt

Timer interrupt handler is modified as follows:

The handler must check whether the current process is the swapper daemon (https://exposnitc.github.io/os_design-files/misc.html#swapper). This condition can happen only when a swap operation is to be initiated. In this case, PAGING_STATUS field of the system status table must be checked and Swap_in/Swap_out function of the pager module must be invoked appropriately (SWAP_OUT = 1 and SWAP_IN = 2).

If the current process is the idle process, there are two possibilities. If swapping is ongoing (check PAGING_STATUS), one can infer that Idle was scheduled because the swapper daemon was blocked. In this case, the timer must invoke the scheduler. (The scheduler will run Idle again if the daemon is not unblocked. Otherwise, the daemon will be scheduled.) The second possibility is that swapping was not on-going. This case is not different from the condition to be checked when timer is entered from any process other than the paging process, and will be descibed next.

Generally, when the timer handler is entered from a process when scheduling was not on, the handler must decide whether normal scheduling shall continue or swap-in/swap-out must be inititiated. Swap-in must be initiated if the value of MEM_FREE COUNT in the system status table is below MEM_LOW. Swap-out must be inititated if either a) memory availability is high (MEM_FREE_COUNT value exceeds MEM_HIGH) or b) some swapped process has TICK value exceeding the threshold MAX_TICK.

Another modification in the Timer interrupt is to increment the TICK field in the process table (https://exposnitc.github.io/os_design-files/process_table.html) of every NON-TERMINATED process. When a process is created by the Fork system call, the TICK value of the process is set to 0 in the process table. Each time the system enters the timer interrupt handler, the TICK value of the process is incremented. The TICK value of a process is reset to zero whenever the process is swapped out or swapped in. Thus the tick value of a process that is not swapped out indicates for how long that process had been in memory without being swapped out. Similarly, the tick value of a swapped out process indicates how long the process had been in swapped state. The swap-in/swap-out algorithms will use the value of TICK to determine the process which had been in swapped state (or not swapped state) for the longest time for swapping in (or out).

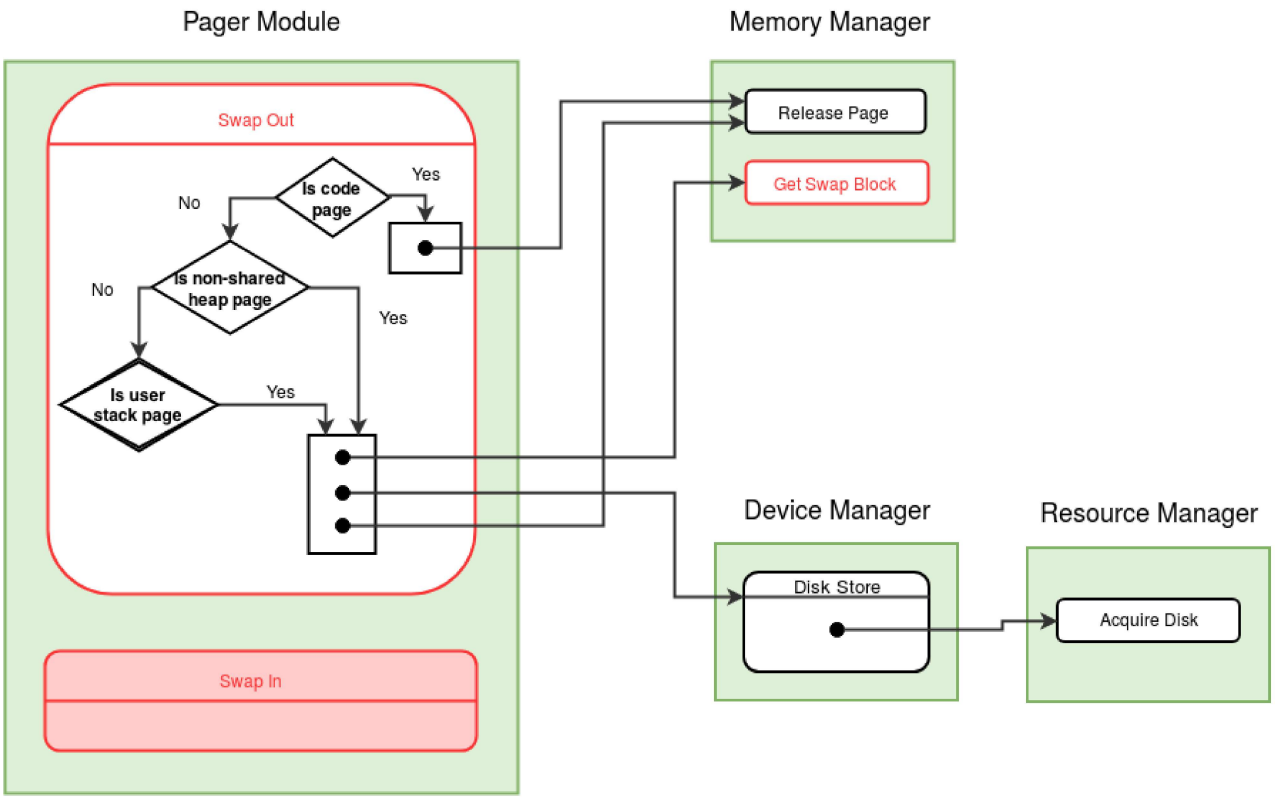
Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

Modify Timer Interrupt implemented in earlier stages according to the detailed algorithm given here (https://exposnitc.github.io/os_design-files/timer.html).

Pager Module (Module 6)

Pager module is responsible for selecting processes to swap-out/swap-in and also to conduct the swap-out/swap-in operations for effective memory management.

1. **Swap Out (function number = 1, Pager module (https://exposnitc.github.io/os_modules/Module_6.html))**



Control flow for **Swap Out**

Swap Out function is invoked from the timer interrupt handler (https://exposnitc.github.io/os_design-files/timer.html) and does not take any arguments. As mentioned earlier, the timer interrupt handler will invoke Swap Out only from the context of the **Swapper Daemon**.

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

Swap Out function first chooses a suitable process for swapping out into the disk. The processes which are not running and are in WAIT_PROCESS or WAIT_SEMAPHORE state are considered first for swapping out (why?). When no such process is found, the process which has stayed longest in the memory is selected for swapping out into the disk. To detect the processes which has stayed longest in the memory, the TICK field in the process table (https://exposnitc.github.io/os_design-files/process_table.html) is used. Thus the process with the highest TICK is selected for swapping out.

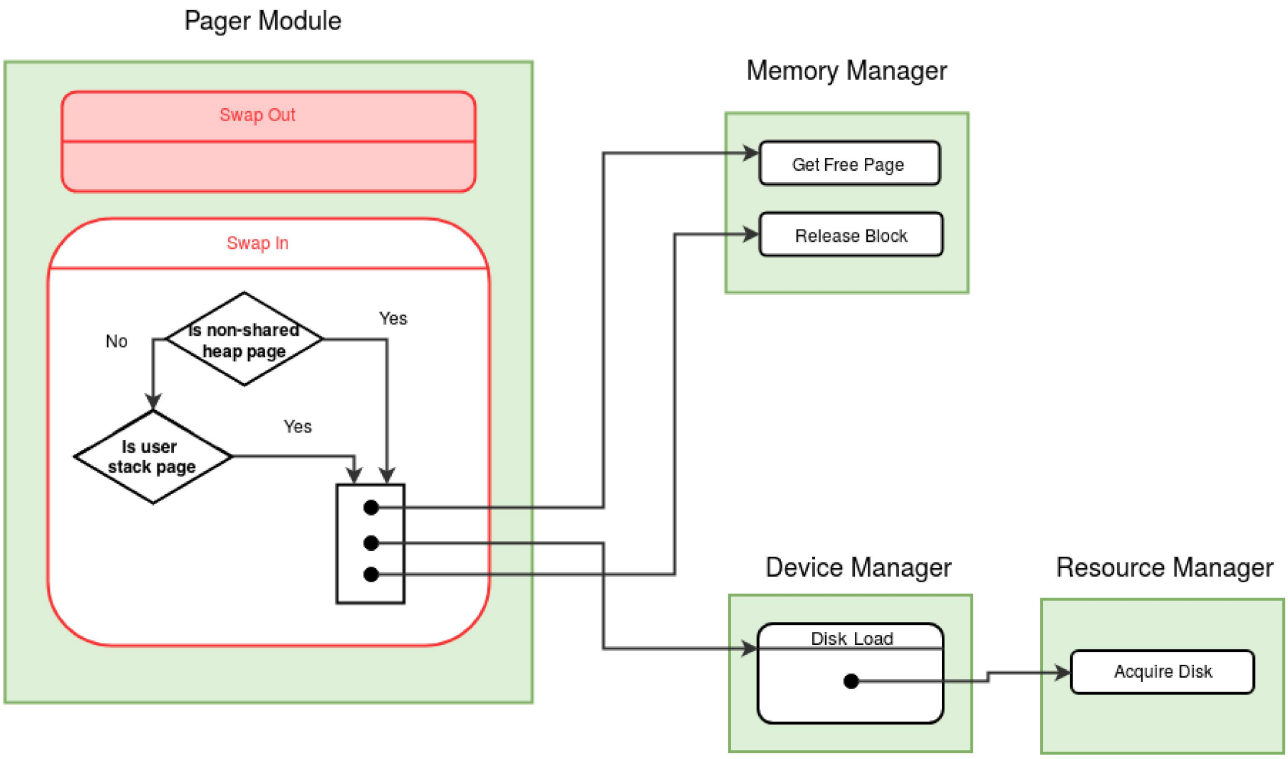
Now that, a process is selected to swap out, the TICK field for the selected process is initialized to 0. (From now on, the TICK field must count for what amount of time the process has been in memory). The code pages for the swapping-out process are released and the page table entries of the code pages are invalidated. The process selected for swapping out, can have shared heap pages. To simplify implementation, shared heap pages are not swapped out into the disk. Again, to simplify implementation, the kernel stack page is also not swapped out. Non-shared heap pages and user stack pages are stored in the swap area in the disk. **Get Swap Block** function of the memory manager module (https://exposnitc.github.io/os_modules/Module_2.html) is invoked to find free blocks in the swap area. These memory pages are stored into the allocated disk blocks by invoking **Disk Store** function of the device manager module (https://exposnitc.github.io/os_modules/Module_4.html) and disk map table (https://exposnitc.github.io/os_design-files/process_table.html#disk_map_table) is updated with the disk numbers of corresponding pages. Memory pages of the process are released using **Release Page** function of memory manager module and page table entries for these swapped out pages are invalidated. Also the SWAP FLAG in the process table of the swapped out process is set to 1, indicating that the process is swapped out.

Finally, the PAGING_STATUS in the System Status Table is reset to 0. This step informs the scheduler that the swap operation is complete and normal scheduling can be resumed.

Implement *Swap Out* function using the detailed algorithm given in the pager module link above.

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

2. Swap In (function number = 2, Pager module (https://exposnitc.github.io/os_modules/Module_6.html))



Control flow for **Swap In**

Swap In function is invoked from the timer interrupt handler (https://exposnitc.github.io/os_design-files/timer.html) and does not take any arguments.

The *Swap In* function selects a swapped out process to be brought back to memory. The process which has stayed for longest time in the disk and is ready to run is selected. That is, the process with the highest TICK among the swapped-out READY processes is selected).

Now that, a process is selected to be swapped back into the memory, the TICK field for the selected process is initialized to 0. Code pages of the process are not loaded into the memory, as these pages can be loaded later when exception occurs during execution of the process. Free memory pages for the heap and user stack are allocated using the **Get Free Page** function of the memory manager module (https://exposnitc.github.io/os_modules/Module_2.html) and disk blocks of the

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

process are loaded into these memory pages using the **Disk Load** function of the device manager module (https://exposnitc.github.io/os_modules/Module_4.html). The Page table is updated for the new heap and user stack pages. The swap disk blocks used by these pages are released using **Release Block** function of the memory manager module and Disk map table (https://exposnitc.github.io/os_design-files/process_table.html#disk_map_table) is invalidated for these pages. Also the SWAP FLAG in the process table of the swapped in process is set to 0, indicating that the process is no longer swapped out.

Finally, the PAGING_STATUS in the System Status Table is reset to 0. This step informs the scheduler that the swap operation is complete and normal scheduling can be resumed.

Implement *Swap In* function using the detailed algorithm given in the pager module link above.

`Note` : **[Implementation Hazard]** There is a possibility that the code of the Pager module will exceed more than 2 disk blocks (more than 512 instructions). Try to write optimized code to fit the pager module code in 2 blocks. You can use the following strategy to reduce the number of instructions. According to given algorithm for **Swap Out** function, the actions done for code pages, heap pages, user stack pages are written separately. This results in calling Release Page function 2 times, Get Swap Block and Disk Store functions 2 times each. Combine these actions into a single while loop where each module function is called only once. The loop should traverse through the page table entries one by one (except library page entries) and perform appropriate actions if the page table entry for a page is valid. Apply similar strategy for **Swap In** function also.

3. **Get Swap Block (function number = 6, Memory Manager Module (https://exposnitc.github.io/os_modules/Module_2.html))**

   Get Swap Block function does not take any arguments. The function returns a free block from the swap area (disk blocks 256 to 511 - see disk organization (https://exposnitc.github.io/os_implementation.html)) of the eXpOS. Get Swap Block searches for a free block from DISK_SWAP_AREA (https://exposnitc.github.io/support_tools-files/constants.html) (starting of disk swap

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

area) to DISK_SIZE (https://exposnitc.github.io/support_tools-files/constants.html)-1 (ending of the eXpOS disk). If a free block is found, the block number is returned. If no free block in swap area is found, -1 is returned to the caller.

Implement *Get Swap Block* function using the detailed algorithm given in the memory manager module link above.

`Note` : The implementation of module functions ***Swap Out***, ***Swap In*** and ***Get Swap Block*** are final.

Modification to Context Switch Module (Module 5)

Previously, the Context Switch module (https://exposnitc.github.io/os_modules/Module_5.html) (scheduler module) would select a new process to schedule according to the Round Robin scheduling algorithm. The procedure for selecting a process to execute is slightly modified in this stage. If swap-in/swap-out is ongoing (that is, if the PAGING_STATUS field of the system status table (https://exposnitc.github.io/os_design-files/mem_ds.html#ss_table) is set), the context switch module schedules the Swapper Daemon (PID = 15) **whenever it is not blocked**. If the swapper daemon is blocked (for some disk operation), then the idle process (PID = 0) must be scheduled. (The OS design disallows scheduling any process except Idle and Swapper daemon when swapping is on-going.) If the PAGING_STATUS is set to 0, swapping is not on-going and hence the next READY/CREATED process which is not swapped out is scheduled in normal Round Robin order. Finally, if no process is in READY or CREATED state, then the idle process is scheduled.

Modify Context Switch module implemented in earlier stages according to the detailed algorithm given here (https://exposnitc.github.io/os_modules/Module_5.html).

Modifications to OS Startup Code

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

Modify OS Startup Code (https://exposnitc.github.io/os_design-files/misc.html#os_startup) to initialize the process table and page table for the Swapper Daemon (similar to the Idle Process).

The final algorithm is given here (https://exposnitc.github.io/os_design-files/misc.html#os_startup).

Modifications to boot module

Modify Boot module (https://exposnitc.github.io/os_modules/Module_7.html) to add the following steps :

- Load module 6 (Pager Module) form disk to memory. See disk/memory organization (https://exposnitc.github.io/os_implementation.html).
- Initialize the SWAPPED_COUNT field to 0 and PAGING_STATUS field to 0 in the system status table (https://exposnitc.github.io/os_design-files/mem_ds.html#ss_table) to 0, as initially there are no swapped out processes.
- Initialize the TICK field to 0 for all the 16 process table (https://exposnitc.github.io/os_design-files/process_table.html) entries.
- Update the MEM_FREE_COUNT to 45 in the system status table (https://exposnitc.github.io/os_design-files/mem_ds.html#ss_table). (The 2 pages are allocated for the user/kernel stack for Swapper Daemon reducing the number from 47 to 45).

Q1. Why only READY state processes are selected for swap in, even though swapped out processes can be in blocked state also? (https://exposnitc.github.io/Roadmap.html#collapseq25)

> It is not very useful to swap in a process which is in blocked state into the memory. As the process is in blocked state, even after swapping in, the process will not execute until it is made READY. Until the process is made READY, it will just occupy memory pages which could be used for some other READY/RUNNING process.

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)

Assignment 1: Write an ExpL program which invokes *Fork* system call four times back to back. Then, the program shall use *Exec* system call to execute pid.xsm file (used in stage 21 (https://exposnitc.github.io/Roadmap.html#collapse21)) to print the PID of the processes. Invoking four Forks back to back is supposed to create 16 new

processes, but only 12 new processes will be created as eXpOS will run out of process table entries. Run this program using the shell in the context of a user.

Assignment 2: Run the program provided here (https://exposnitc.github.io/test_prog.html#test_program_8) using shell in the context of a user. The program given in the given link will first read a delay parameter and then, call the Fork system call and create 12 processes. Each process prints numbers from PID*100 to PID*100 + 7. After printing each number, a delay function is called with the the delay parameter provided.

Assignment 3: Run the program provided here (https://exposnitc.github.io/test_prog.html#test_program_9) using shell in the context of a user. The program will create a file with name as "num.dat" and permission as *open access*. Integers 1 to 1200 are written to this file and file is closed. The program will then invoke *Fork* system call four times, back to back to create 12 processes and *Exec* system call is invoked with file "pgm1.xsm". The program for "pgm1.xsm" is provided here (https://exposnitc.github.io/test_prog.html#test_program_10). "pgm1.xsm" will create a new file according to the PID of the process and read 100 numbers from file "num.dat" from offset (PID-3)*100 to (PID-3)*100+99 and write to newly created file. After successful execution, there should be 12 data files each containing 100 consecutive numbers (PID-3)*100+1 to (PID-3)*100+100.

Assignment 4: Run the program provided here (https://exposnitc.github.io/test_prog.html#test_program_11) using shell in the context of a user. The program will create a file with name as "numbers.dat" and permission as *open access* and open the file. The program also invokes *Semget* for a shared semaphore. The program will then invoke *Fork* system call four times, back to back to create 12 processes. The 12 processes now share a file open instance and a semaphore. Each process will write 100 numbers consecutively (PID*1000+1 to PID*1000+100) to the file "numbers.dat". Each process then invokes the *Exec* system call to run the program "pgm2.xsm". The program for "pgm2.xsm" is provided here (https://exposnitc.github.io/test_prog.html#test_program_12). "pgm2.xsm" will create a new file according to the PID of the process and read 100 numbers from file "numbers.dat" from offset (PID-3)*100 to (PID-3)*100+99 and write to newly created file. After successful execution, there should be 12 data files each containing 100 numbers from X*1000 to X*1000+99, where X ∈ {3,4..15}. The numbers written by a process in the newly created file need not be the same numbers the process has written in "numbers.dat" file.

Assignment 5: Run the program *(merge.expl)* provided here (https://exposnitc.github.io/test_prog.html#test_program_16) using shell in the context of a user. The *merge.expl* program, first stores numbers from 1 to 512 in a random order into a file *merge.dat*. It then forks and executes *m_store.expl* which creates 8 files *temp{i}.dat*, where i=1..8 and stores 64 numbers each from *merge.expl*.

Top (https://exposnitc.github.io/Roadmap.html#navtop)

Then, all the temporary files are sorted by executing *m_sort.expl*. Next, the first ExpL program forks and executes *m_merge.expl* which merges all the temporary files back into *merge.dat* and finally, prints the contents of the file in ascending order. If all the system calls in your OS implementation works correctly, then numbers 1 to 512 will be printed out in order.

`Note` : To run the program provided by the user, Shell process first invokes *fork* to create a child process. Shell will wait until this first child process completes its execution. When the first child exits, shell will resume execution even if some processes created by the given program are running in the background. This can lead to the following interesting situation. Suppose that all active processes execept idle, login and shell were swapped out and the XSM simulator is waiting for terminal input from the user into the shell. In this case, you will have to issue some command to the shell, so that the system keeps on running.

Close (https://exposnitc.github.io/Roadmap.html#collapse27)

Stage 28 : Multi-Core Extension (12 Hours) (https://exposnitc.github.io/Roadmap.html#collapse28)

Top ↑ (https://exposnitc.github.io/Roadmap.html#navtop)