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)

Learning Objectives (https://exposnitc.github.io/Roadmap.html#lo22)
Understanding how semaphores help to solve the critical section problem.
Add support for semaphores to eXpOS.

Pre-requisite Reading (https://exposnitc.github.io/Roadmap.html#lo22a)
Read and understand Resource Sharing (https://exposnitc.github.io/os_spec-files/expos_abstractions.html#resource_sharing) and Access Control (https://exposnitc.github.io/os_spec-files/synchronization.html#access_control) documentations of eXpOS before proceeding further.

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

In this stage, we will add support for semaphores (https://en.wikipedia.org/wiki/Semaphore_%28programming%29) to the OS. Semaphores are primitives that allow concurrent processes to handle the critical section (https://en.wikipedia.org/wiki/Critical_section) problem. A typical instance of the critical section problem occurs when a set of processes share memory or files. Here it is likely to be necessary to ensure that the processes do not access the shared

data (or file) simultaneously to ensure data consistency. eXpOS provides **binary semaphores** which can be used by user programs (ExpL programs) to synchronize the access to the shared resources so that data inconsistency will not occur.

There are four actions related to semaphores that a process can perform. Below are the actions along with the corresponding eXpOS system calls -

1) Acquiring a semaphore - *Semget* system call
2) Releasing a semaphore - *Semrelease* system call
3) Locking a semaphore - *SemLock* system call
4) Unlocking a semaphore - *SemUnLock* system call

To use a semaphore, first a process has to acquire a semaphore. **When a process forks, the semaphores currently acquired by a process is shared between the child and the parent.** A process can lock and unlock a semaphore only after acquiring the semaphore. The process can lock the semaphore when it needs to enter into the critical section. After exiting from the critical section, the process unlocks the semaphore allowing other processes (with which the semaphore is shared) to enter the critical section. After the use of a semaphore is finished, a process can detach the semaphore by releasing the semaphore.

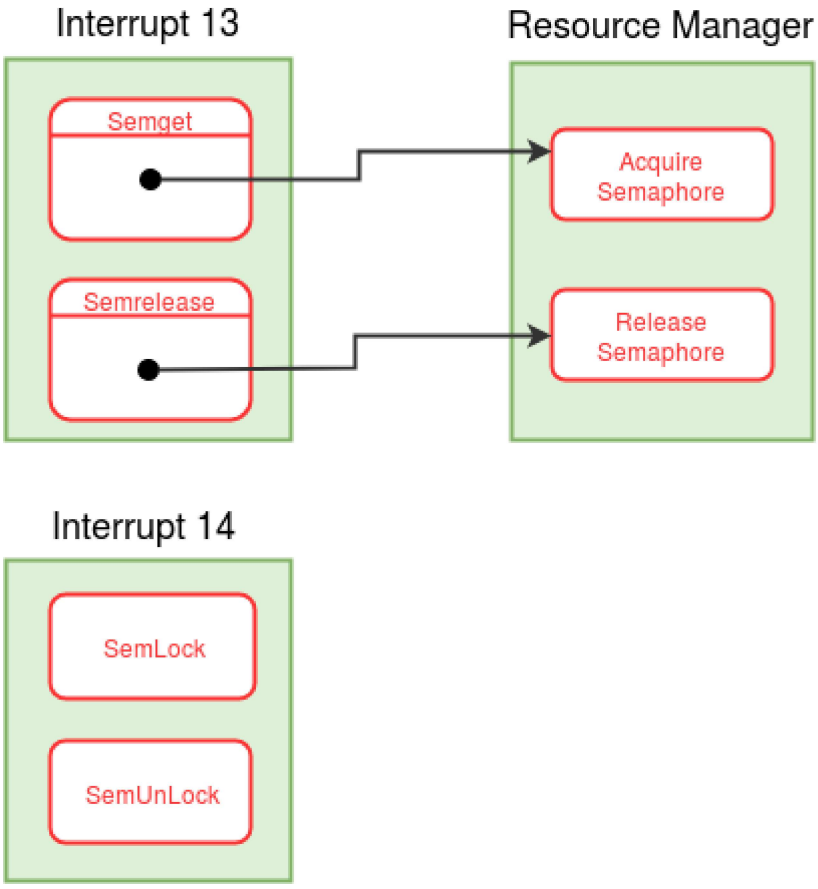
A process maintains record of the semaphores acquired by it in its per-process resource table (https://exposnitc.github.io/os_design-files/process_table.html#per_process_table). eXpOS uses the data structure, semaphore table (https://exposnitc.github.io/os_design-files/mem_ds.html#sem_table) to manage semaphores. Semaphore table is a global data structure which is used to store details of semaphores currently used by all the processes. The Semaphore table has 32 (MAX_SEM_COUNT (https://exposnitc.github.io/support_tools-files/constants.html)) entries. This means that only 32 semaphores can be used by all the processes in the system at a time. Each entry in the semaphore table occupies four words of which the last two are currently unused. For each semaphore, the PROCESS COUNT field in it's semaphore table entry keeps track of the number of processes currently sharing the semaphore. If a process locks the semaphore, the LOCKING PID field is set to the PID of that process. LOCKING PID is set to -1 when the semaphore is not locked by any process. An invalid semaphore table entry is indicated by PROCESS COUNT equal to 0. The SPL constant SEMAPHORE_TABLE (https://exposnitc.github.io/support_tools-

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

files/constants.html) gives the starting address of the semaphore table in the memory (https://exposnitc.github.io/os_implementation.html). See semaphore table (https://exposnitc.github.io/os_design-files/mem_ds.html#sem_table) for more details.

The per-process resource table (https://exposnitc.github.io/os_design-files/process_table.html#per_process_table) of each process keeps track of the resources (semaphores and files) currently used by the process. The per-process resource table is stored in the last 16 words of the user area page (https://exposnitc.github.io/os_design-files/process_table.html#user_area) of a process. Per-process resource table can store details of at most eight resources at a time. Hence the total number of semaphores and files acquired by a process at a time is at most eight. Each per process resource table entry contains two words. The first field, called the **Resource Identifier** field, indicates whether the entry corresponds to a file or a semaphore. For representing the resource as a file, the SPL constant FILE (https://exposnitc.github.io/support_tools-files/constants.html) (0) is used and for semaphore, the SPL constant SEMAPHORE (https://exposnitc.github.io/support_tools-files/constants.html) (1) is used. The second field stores the index of the semaphore table entry if the resource is a semaphore. (If the resource is a file, an index to the open file table entry will be stored - we will see this in later stages.) See the description of per-process resource table (https://exposnitc.github.io/os_design-files/process_table.html#per_process_table) for details.

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



Control flow for *Semaphore* system calls

Implementation of Interrupt routine 13

The system calls *Semget* and *Semrelease* are implemented in the interrupt routine 13. *Semget* and *Semrelease* has system call numbers 17 and 18 respectively.

- Extract the system call number from the user stack and switch to the kernel stack.
- Implement system calls *Semget* and *Semrelease* according to the system call number extracted from above step. Steps to implement these system calls are explained below.
- Change back to the user stack and return to the user mode.

1. Semget System Call

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

Semget system call is used to acquire a new semaphore. *Semget* finds a free entry in the per-process resource table (https://exposnitc.github.io/os_design-files/process_table.html#per_process_table). *Semget* then creates a new entry in the

semaphore table by invoking the **Acquire Semaphore** function of resource manager module (https://exposnitc.github.io/os_modules/Module_0.html). The index of the semaphore table entry returned by Acquire Semaphore function is stored in the free entry of per-process resource table of the process. Finally, *Semget* system call returns the index of newly created entry in the per-process resource table as **semaphore descriptor** (SEMID).

Implement *Semget* system call using the detailed algorithm provided here (https://exposnitc.github.io/os_design-files/semaphore_algos.html#semget).

2. **Semrelease System Call**

Semrelease system call takes semaphore desciptor (SEMID) as argument from user program.*Semrelease* system call is used to detach a semaphore from the process. *Semrelease* releases the acquired semaphore and wakes up all the processes waiting for the semaphore by invoking the **Release Semaphore** function of resource manager module (https://exposnitc.github.io/os_modules/Module_0.html). *Semrelease* also invalidates the per-process resource table entry corresponding to the SEMID given as an argument.

Implement *Semrelease* system call using the detailed algorithm provided here (https://exposnitc.github.io/os_design-files/semaphore_algos.html#semrelease).

`Note` : If any semaphore is not released by a process during execution using *Semrelease* system call, then the semaphore is released at the time of termination of the process in *Exit* system call.

3. **Acquire Semaphore (function number = 6, resource manager module (https://exposnitc.github.io/os_modules/Module_0.html))**

Acquire Semaphore function takes PID of the current process as argument. *Acquire Semaphore* finds a free entry in the semaphore table and sets the PROCESS COUNT to 1 in that entry. Finally, *Acquire Semaphore* returns the index of that free entry of semaphore table.

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

Implement *Acquire Semaphore* function using the detailed algorithm provided in resource manager module link above.

4. **Release Semaphore (function number = 7, resource manager module (https://exposnitc.github.io/os_modules/Module_0.html))**

   Release Semaphore function takes a semaphore index (SEMID) and PID of a process as arguments. If the semaphore to be released is locked by current process, then *Release Semaphore* function unlocks the semaphore and wakes up all the processes waiting for this semaphore. *Release Semaphore* function finally decrements the PROCESS COUNT of the semaphore in its corresponding semaphore table entry.

   Implement *Release Semaphore* function using the detailed algorithm provided in resource manager module link above.

Implementation of Interrupt routine 14

The system calls *SemLock* and *SemUnLock* are implemented in the interrupt routine 14. *SemLock* and *SemUnLock* has system call numbers 19 and 20 respectively.

- Extract the system call number from the user stack and switch to the kernel stack.
- Implement system calls *SemLock* and *SemUnLock* according to the system call number extracted from above step. Steps to implement these system calls are explained below.
- Change back to the user stack and return to the user mode.

1. **SemLock System Call**

   SemLock system call takes a semaphore desciptor (SEMID) as an argument from user program. A process locks the semaphore it is sharing using the *SemLock* system call. If the requested semaphore is currently locked by some other process, the current process blocks its execution by changing its STATE (https://exposnitc.github.io/os_design-files/process_table.html#state) to the tuple (WAIT_SEMAPHORE, semaphore table index of requested semaphore) until the requested semaphore is unlocked. When the semaphore is unlocked, then STATE of the current process is made READY (by the process which has unlocked the semaphore). When the current process is scheduled and the semaphore is still unlocked the current process locks the semaphore by changing the LOCKING PID in

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

the semaphore table entry to the PID of the current process. When the process is scheduled but finds that the semaphore is locked by some other process, current process again waits in the busy loop until the requested semaphore is unlocked.

Implement *SemLock* system call using the detailed algorithm provided here (https://exposnitc.github.io/os_design-files/semaphore_algos.html#semlock).

2. **SemUnLock System Call**

SemUnLock system call takes a semaphore desciptor (SEMID) as argument. A process invokes *SemUnLock* system call to unlock the semaphore. *SemUnLock* invalidates the LOCKING PID field (store -1) in the semaphore table entry for the semaphore. All the processes waiting for the semaphore are made READY for execution.

Implement *SemUnLock* system call using the detailed algorithm provided here (https://exposnitc.github.io/os_design-files/semaphore_algos.html#semunlock).

`Note` : The implementation of ***Semget***, ***Semrelease***, ***SemLock***, ***SemUnLock*** system calls and **Acquire Semaphore**, **Release Semaphore** module functions are final.

Modifications to *Fork* system call

In this stage, *Fork* is modified to update the semaphore table for the semaphores acquired by the parent process. When a process forks, the semaphores acquired by the parent process are now shared between parent and child. To reflect this change, PROCESS COUNT field is incremented by one in the semaphore table entry for every semphore shared between parent and child. Refer algorithm for fork system call (https://exposnitc.github.io/os_design-files/fork.html).

- While copying the per-process resource table of parent to the child process do following -
- If the resource is semaphore (check the Resource Identifier field in the per-process resource table (https://exposnitc.github.io/os_design-files/process_table.html#per_process_table)), then using the sempahore table index, increment the PROCESS COUNT field in the semaphore table (https://exposnitc.github.io/os_design-files/mem_ds.html#sem_table) entry.

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

Modifications to Free User Area Page (function number = 2, process manager module (https://exposnitc.github.io/os_modules/Module_1.html))

The user area page of every process contains the per-process resource table (https://exposnitc.github.io/os_design-files/process_table.html#per_process_table) in the last 16 words. When a process terminates, all the semaphores the process has acquired (and haven't released explicitly) have to be released. This is done in the *Free User Area Page* function. The **Release Semaphore** function of resource manager module is invoked for every valid semaphore in the per-process resource table of the process.

- For each entry in the per-process resource table of the process do following -
- If the resource is valid and is semaphore (check the Resource Identifier field in the per-process resource table (https://exposnitc.github.io/os_design-files/process_table.html#per_process_table)), then invoke **Release Semaphore** function of resource manager module (https://exposnitc.github.io/os_modules/Module_0.html).

`Note` : ***Fork*** system call and **Free User Area page** function will be further modified in later stages for the file resources.

Modifications to boot module

- Initialize the semaphore table (https://exposnitc.github.io/os_design-files/mem_ds.html#sem_table) by setting PROCESS COUNT to 0 and LOCKING PID to -1 for all entries.
- Load interrupt routine 13 and 14 from the disk to the memory. See memory organisation (https://exposnitc.github.io/os_implementation.html).

Making things work

Compile and load the newly written/modified files to the disk using XFS-interface.

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

Q1. When a process waiting for a sempahore is scheduled again after the sempahore is unlocked, is it possible that the process finds the sempahore still locked? (https://exposnitc.github.io/Roadmap.html#collapseq20)

> Yes, it is possible. As some other process waiting for the semaphore could be scheduled before the current process and could have locked the semaphore. In this case the present process finds the semaphore locked again and has to wait in a busy loop until the required sempahore is unlocked.

Q2.A process first locks a semaphore using SemLock system call and then forks to create a child. As the semaphore is now shared between child and parent, what will be locking status for the semaphore? (https://exposnitc.github.io/Roadmap.html#collapseq21)

> The sempahore will still be locked by the parent process. In *Fork* system call, the PROCESS COUNT in the semaphore table is incremented by one but LOCKING PID field is left untouched.

Assignment 1: The reader-writer program given here (https://exposnitc.github.io/test_prog.html#test_program_4) has two writers and one reader. The parent process will create two child processes by invoking *fork*. The parent and two child processes share a buffer of one word. At a time only one process can read/write to this buffer. To acheive this, these three processes use a shared semaphore. A writer process can write to the buffer if it is empty and the reader process can only read from the buffer if it is full. Before the word in the buffer is overwritten the reader process must read it and print the word to the console. The parent process is the reader process and its two children are writers. One child process writes even numbers from 1 to 100 and other one writes odd numbers from 1 to 100 to the buffer. The parent process reads the numbers and prints them on to the console. Compile the program given in link above and execute the program using the shell. The program must print all numbers from 1 to 100, but not necessarily in sequential order.

Assignment 2: The ExpL programs given here (https://exposnitc.github.io/test_prog.html#test_program_13) describes a *parent.expl* program and a *child.expl* program. The *parent.xsm* program will create 8 child processes by invoking *Fork* 3 times. Each of the child processes will print the process ID (PID) and then, invokes the *Exec* system call to execute the program *"child.xsm"*. The *child.xsm* program stores numbers from *PID*100* to *PID*100 + 9* onto a linked list

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

and prints them to the console (each child process will have a seperate heap as the Exec system call alocates a seperate heap for each process). Compile the programs given in the link above and execute the parent program (*parent.xsm*) using the shell. The program must print all numbers from *PID*100* to *PID*100+9*, where PID = 2 to 9, but not necessarily in sequential order. Also, calculate the **maximum memory usage, number of disk access and number of context switches** (by modifying the OS Kernel code).

Assignment 3: The two ExpL programs given here (https://exposnitc.github.io/test_prog.html#test_program_14) perform merge sort in two different ways. The first one is done in a sequential manner and the second one, in a concurrent approach. Values from 1 to 64 are stored in decreasing order in a linked list and are sorted using a recursive merge sort function. In the concurrent approach, the process is forked and the merge sort function is called recursively for the two sub-lists from the two child processes. Compile the programs given in the link above and execute each of them using the shell. The program must print values from 1 to 64 in a sorted manner. Also, calculate the **maximum memory usage, number of contexts switches and the number of switches to KERNEL mode**.

Close (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)

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

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

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