

✓ Stage 15 : Resource Manager Module (4 Hours)

✓ Stage 16 : Console Input (6 Hours)

✓ Stage 17 : Program Loader (6 Hours)

Learning Objectives

- 👉 Familiarize with Process manager and Memory Manager modules.
- 👉 Enable the OS to load and execute application programs from the disk (exec system call).

Pre-requisite Reading

- 👉 Before proceeding further review the XEXE file format and the address space model of a process from the eXpOS ABI documentation (abi.html).
- 👉 Make sure to be thorough with the SPL module calling conventions. (support_tools-files/spl.html)

1
2
3


In this stage, you will be working on the implementation of the exec system call. Exec is the "program loader" of eXpOS. Exec takes as input a filename. It first checks whether the file is a **valid eXpOS executable** stored in the XSM disk, **adhering to the XEXE format** (abi.html#xexe). If so, Exec destroys the invoking process, loads the executable file from the disk, and sets up the program for execution as a process. A **successful exec operation results in the termination of the invoking process and hence never returns to it.**

Name of the executable file is the only input to the exec system call. This file should be present in the disk before starting the machine. The inode index of this file can be obtained by going through the memory copy of the inode table (os_design-files/disk_ds.html).

Note : From here onwards **whenever the inode table is referred, it is implied that memory copy of the inode table is referred** (unless mentioned otherwise). Inode table is loaded from the disk to the memory in Boot Module. Top ↑

Exec first deallocates all pages the invoking process is using. These include two heap pages, two user stack pages, code pages occupied by the process and the user area page. It also invalidates the entries of the page table of the invoking process. Note that the **newly scheduled process will have the same PID as that of the invoking process**. Hence the same process table entry and page table of the invoking process will be used by the newly loaded process. Exec calls the **Exit Process** function in the process manager module (os_modules/Module_1.html) (module 1) to deallocate the pages and to terminate the current process.

1=
2=
3=

As mentioned earlier, Exit Process function releases the user area page of the current process. **Since Exec system call runs in kernel mode and needs a kernel stack for its own execution, after coming back from Exit process function, exec reclaims the same user area page for the new process.** Further, exec acquires two heap, two stack and the required number of code pages (number of disk blocks in the inode entry of the file in the inode table) for the new process. New pages will be acquired by invoking the **Get Free Page** function present in the memory manager module (os_modules/Module_2.html) (module 2). Page table is updated according to the new pages acquired. Code blocks for the new process are loaded from the disk to the memory pages obtained. For loading blocks into the memory pages, we use  immediate load (loadi statement (support_tools-files/spl.html) in SPL). Finally, exec initializes the IP value of new process on top of its user stack and initiates execution of the newly loaded process in the user mode.

eXpOS maintains a data structure called memory free list (os_design-files/mem_ds.html#mem_free_list) in page 57 of the memory (os_implementation.html). Each Page can be shared by zero or more processes. There are 128 entries in the memory free list corresponding to each page of memory. **For each page, the corresponding entry in the list stores the number of processes sharing the page.** The constant MEMORY_FREE_LIST (support_tools-files/constants.html) gives the starting address of the memory free list.

Now we will understand the working of the module functions that exec uses during its execution. Before proceeding further have a careful look of the diagram to understand the overall working of the exec system call.

Top ↑

1. Exit Process (function number = 3, process manager module (os_modules/Module_1.html))

The first function invoked in the exec system call is the **Exit Process** function. Exit Process function takes PID of a process as an argument (In this stage, PID of the current process is passed). Exit process deallocates all the pages of the invoked process. It deallocates the pages present in the page table by invoking another function called **Free Page Table** present in the same module. Further, The Exit Process deallocates the user area page by invoking **Free User Area Page** in the same module. The state of the process (corresponding to the given PID) is set to TERMINATED. This is not the final Exit process function. There will be minor modifications to this function in the later stages.

2. Free Page Table (function number = 4, process manager module (os_modules/Module_1.html))

The Free Page Table function takes PID of a process as an argument (In this stage, PID of the current process is passed). In the function Free Page Table, for every valid entry in the page table of the process, the corresponding page is freed by invoking the **Release Page** function present in the memory manager module (os_modules/Module_2.html). Since the library pages are shared by all the processes, do not invoke the Release Page function for the library pages. Free Page Table function invalidates all the page table entries of the process with given PID. The part of the Free Page Table function involving updates to the Disk Map Table will be taken care in subsequent stages.

3. Free User Area Page (function number = 2, process manager module (os_modules/Module_1.html))

The function **Free User Area Page** takes PID of a process (In this stage, PID of the current process is passed) as an argument. The user area page number of the process is obtained from the process table entry corresponding to the PID. This user area page is freed by invoking the **Release Page** function from the memory manager module. However, since we are using Free User Area Page to release the user area page of the current process one needs to be careful here. The user area page holds the kernel stack of the current process. Hence, releasing this page means that the page holding the return address for the call to Free User Area Page function itself has

Top ↑

been released! Nevertheless the return address and the saved context of the calling process will not be lost. This is because Release Page function is non blocking and hence the page will never be allocated to another process before control transfers back to the caller. (Free User Area Page function also releases the resources like files and semaphores acquired by the process. We will look into it in later stages.)

4. Release Page (function number = 2, memory manager module (os_modules/Module_2.html))

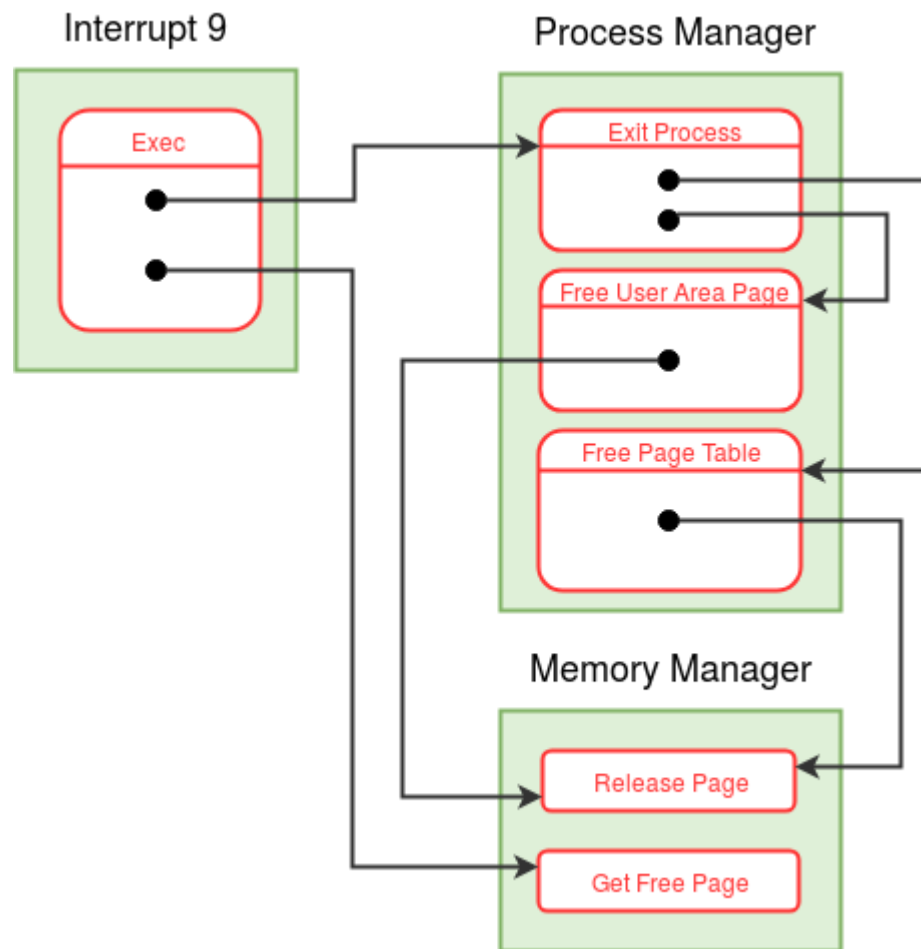
1=
2=
3=

The Release Page function takes the page number to be released as an argument. The Release Page function decrements the value in the memory free list corresponding to the page number given as an argument. Note that we don't tamper with the content of the page as the page may be shared by other processes. The system status table (os_design-files/mem_ds.html#ss_table) keeps track of number of free memory pages available to use in the MEM_FREE_COUNT field. When the memory free list entry of the page becomes zero, no process is currently using the page. In this case, increment the value of MEM_FREE_COUNT in the system status table indicating that the page has become free for fresh allocation. Finally, Release page function must check whether there are processes in WAIT_MEM state (these are processes blocked, waiting for memory to become available). If so, these processes have to be set to READY state as memory has become available now. At present, the OS does not set any process to WAIT_MEM state and this check is superfluous. However, in later stages, the OS will set processes to WAIT_MEM state when memory is temporarily unavailable, and hence when memory is released by some process, the waiting processes have to be set to READY state.

5. Get Free Page (function number = 1, memory manager module (os_modules/Module_2.html))

To acquire the pages for the new process, exec calls the module function Get Free Page. The function returns the page number of the page allocated. Fundamentally, Get Free page searches through the memory free list to find a free page for the allocation. If a free page is found, memory free list entry corresponding to that page is incremented and number of the page found is returned.

If no memory page is free (MEM_FREE_COUNT in the system status table is 0), then state of the process is changed to WAIT_MEM and the scheduler is invoked. This process is scheduled again, when the memory is available (state of this process is changed to READY by some other process). The field WAIT_MEM_COUNT in the system status table stores the number of processes waiting to acquire a memory page. The Get Free Page function increments the WAIT_MEM_COUNT before changing state to WAIT_MEM. The process waits in the WAIT_MEM state until any memory page is available for use. Upon waking up, the Get Free Page function allocates the free memory page and updates the WAIT_MEM_COUNT and MEM_FREE_COUNT in the system status table.



Control flow for *Exec* system call

Implementation of Exec system call

Top ↑

Exec (os_design-files/Sw_interface.html) has system call number as 9 and it is implemented in interrupt routine 9. Follow steps below to implement interrupt routine 9.

1. Save user stack value for later use, set up the kernel stack. (see kernel stack management during system calls (os_design-files/stack_smcall.html).)
2. Set the MODE FLAG in the process table (os_design-files/process_table.html) to system call number of exec.
3. Get the argument (name of the file) from user stack.
4. Search the memory copy of the inode table (os_design-files/disk_ds.html#inode_table) for the file, If the file is not present or file is not in XEXE format return to user mode with return value -1 indicating failure (after setting up MODE FLAG and the user stack).
5. If the file is present, save the inode index of the file into a register for future use.
6. Call the **Exit Process** function in process manager module (os_modules/Module_1.html) to deallocate the resources and pages of the current process.
7. Get the user area page number from the process table of the current process. This page has been deallocated by the Exit Process function. Reclaim the same page by incrementing the memory free list entry of user area page and decrementing the MEM_FREE_COUNT field in the system status table (os_design-files/mem_ds.html#ss_table). (same user area page is reclaimed - why?)
8. Set the SP to the start of the user area page to initialize the kernel stack of the new process.
9. New process uses the PID of the terminated process. Update the STATE field to RUNNING and store inode index obtained above in the inode index field in the process table.
10. Allocate new pages and set the page table entries for the new process.
 - i. Set the library page entries in the page table. (must be set to read only-why? Note that library page need not be allocated.)
 - ii. Invoke the **Get Free Page** function to allocate 2 stack and 2 heap pages. Also validate the corresponding entries in page table.
 - iii. Find out the number of blocks occupied by the file from inode table (os_design-files/disk_ds.html#inode_table). Allocate same number of code pages by invoking

1=
2=
3=

the **Get Free Page** function and update the page table entries.

11. Load the code blocks from the disk to the memory pages using loadi statement (support_tools-files/spl.html). (We will change this step in the next stage.)
12. Store the entry point IP (present in the header of first code page) value on top of the user stack.
13. Change SP to user stack, change the MODE FLAG back to user mode and return to user mode.

1
2
3

Note: The implementation of some of the module functions given below are primitive versions of their final versions. The present versions are just sufficient for the purposes of this stage. These functions may require modifications in later stages. The required modifications will be explained at appropriate points in the roadmap.

Implementation of Process Manager Module (os_modules/Module_1.html) (Module 1)

1. According to the function number value present in R1, implement different functions in module 1.
2. If the function number corresponds to **Free User Area Page**, follow steps below
 - i. Obtain the user area page number from the process table (os_design-files/process_table.html) entry corresponding to the PID given as an argument.
 - ii. Free the user area page by invoking the **Release Page** function.
 - iii. Return to the caller.
3. If the function number corresponds to **Exit Process**, follow steps below
 - i. Extract PID of the invoking process from the corresponding register.
 - ii. Invoke the **Free Page Table** function with same PID to deallocate the page table entries.
 - iii. Invoke the **Free user Area Page** function with the same PID to free the user area page.
 - iv. Set the state of the process as TERMINATED and return to the caller.
4. If the function number corresponds to **Free Page Table**, follow steps below
 - i. Invalidate the page table entries for the library pages by setting page number as -1 and auxiliary data as "0000" for each entry.

Top ↑

- ii. For each valid entry in the page table, release the page by invoking the **Release Page** function and invalidate the entry.
- iii. Return to the Caller.

Note : The implementation of above three functions of process manager module are not final versions. They will be updated as required in later stages.

Implementation of Memory Manager Module (os_modules/Module_2.html) (Module 2)

1. According to the function number value present in R1, implement different functions in module 2.
2. If the function number corresponds to **Get Free Page**, follow steps below
 - i. Increment WAIT_MEM_COUNT field in the system status table. //Do not increment the WAIT_MEM_COUNT in busy loop (an important step)
 - ii. While memory is full (MEM_FREE_COUNT will be 0), do following.
 - Set the state of the invoked process as WAIT_MEM.
 - Schedule other process by invoking the context switch module (os_modules/Module_5.html). // blocking the process
 - iii. Decrement the WAIT_MEM_COUNT field and MEM_FREE_COUNT field in the system status table.

/* Note the sequence - increment WAIT_MEM_COUNT, waiting for the memory, decrement WAIT_MEM_COUNT.*/
 - iv. Find a free page using memory free list and set the corresponding entry as 1. Make sure to store the obtained free page number in R0 as return value.
 - v. Return to the caller.
3. If the function number corresponds to **Release Page**, follow steps below
 - i. The Page number to be released is present in R2. Decrement the corresponding entry in the memory free list.
 - ii. If that entry in the memory free list becomes zero, then the page is free. So increment the MEM_FREE_COUNT in the system status table. Top ↑
 - iii. Update the STATUS to READY for all processes (with valid PID) which have STATUS as WAIT_MEM.

iv. Return to the caller.

Note : The Get Free Page and Release Page functions implemented above are final versions according to the algorithm given in memory manager module (os_modules/Module_2.html).

Modifications to the Boot Module.

1. Load interrupt routine 9, module 1, module 2 and inode table from the disk to the memory. Refer to disk and memory organization here (os_implementation.html).
2. Initialize the memory free list with value 1 for pages used and 0 for free pages.
3. Initialize the fields WAIT_MEM_COUNT to 0 and MEM_FREE_COUNT to number of free pages in the system status table.

ExpOS memory layout for the XSM machine sets pages 76-127 as free pages and the remaining are reserved for OS modules, OS data structures, interrupts, system calls, code region of special processes (INIT, shell), the OS library etc. Hence, the initial setting of the memory free list should mark pages 76-127 as free (value 0) and the rest as allocated (value 1). The memory manager shall be doing allocation and deallocation only from the free pool (76-127). The boot module/OS startup code further allocates space for the user-area page and stack pages for INIT and IDLE processes from this free pool. The INIT process require two additional pages for the heap. The pages allocated to INIT process are- stack-76, 77, heap-78, 79, user area page-80 and for IDLE process- stack-81, user area page-82. Hence, the effective free pool (MEM_FREE_COUNT value) at the end of OS initialization process starts from memory page 83.

Making things work

Top ↑

1. Compile and load interrupt routine 9, module 1 and module 2 using XFS-interface. Run the machine.

Q1. Why does exec reclaim the same user area page for the new process? (As done in step 7 of exec system call implementation.)

Since the page storing the kernel context has been de-allocated, before making any function call, a stack page has to be allocated to store parameters, return address etc. It is unsafe to invoke the Get Free Page function of the memory manager module before allocating a stack page (why?).

1
2
3

Q2. Why should the OS set the WRITE PERMISSION BIT for library and code pages in each page table entry to 0, denying permission for the process to write to these pages?

ExpOS does not expect processes to modify the code page during execution. Hence, during a fork() system call (to be seen in later stages), the code pages are shared between several processes. Similarly, the library pages are shared by all processes. If a process is allowed to write into a code/library page, the shared program/library will get modified and will alter the execution behaviour of other processes, which violates the basic virtual address space (https://en.wikipedia.org/wiki/Virtual_address_space) model offered by the OS.

Assignment 1: [Shell version-I] Write an ExpL program to read the name of program from the console and execute that program using exec system call. Load this program as INIT program and run the odd.expl (printing odd numbers between 1-100) program using the shell.

Assignment 2: Use the XSM debugger (support_tools-files/xsm-simulator.html) to print out the contents of the System Status Table and the Memory Free List after Get Free Page and Release Page functions, inside the Memory Manager module.

Top ↑

✕ Close

✓ Stage 18 : Disk Interrupt Handler (6 Hours)