Enhancement to XOS Operating System and XFS File System

Kruthika Suresh Ved B110300CS
Sikha V Manoj B110572CS
Sonia V Mathew B110495CS
Guided by: Dr.K.Muralikrishnan

March 3, 2015

Abstract

Project eXpOS or experimental Operating System is a educational platform to develop an operating system. It is an instructional tool for students to learn and implement OS data structures and functionalities on a simulated machine called XSM (eXperimental String Machine).The OS is programmed using a custom language known as SPL (System Programmer's Language) and application programs, which run on the OS, are programmed using eXpL (Experimental Programmer's Language).

1 Problem Definition

This project aims to extend XOS with features like Shared Memory Model, InterProcess Communication, Synchronization and Directory Structure.

2 eXpOS Specification

eXpOS has a very simple specification that allows a junior undergraduate computer science student to implement it in a few months, subject to availability of adequate hardware and programming platform support. This OS specification is prepared in a manner independent of programming language and target machine.

2.1 eXperimental File System (eXpFS)

eXpOS uses eXpFS (eXperimental File System) which contains files organized into a single directory called the root. There are three types of eXpFS files: the root, data files and executable files. The root is also treated conceptually as a file.

2.1.1 Root

The root file has name **root** and contains information about the files stored in the file system. For each file stored in eXpFS, the root stores three words of information: file-name, file-size and file-type. This triple is called the root entry for the file. The first root entry is for the root itself.

2.1.2 Data File

A data file is a sequence of words. eXpFS expects the Operating System to display data files with an extension .dat. eXpFS treats this as a default file type, hence the application programs do not have to specify the extension .dat at the time of file creation. The operations allowed in data files are Create, Delete, Open, Close, FLock, FUnlock, Read, Write, Seek.

2.1.3 Executable File

Executable files are essentially program files that must be loaded and run by the operating system. The executable file format recognized by eXpOS is called the Experimental executable file (XEXE) format. In this format, an executable file is divided into two sections - Header and Code (In this implementation of eXpOS, static data is stored in stack pages).

2.2 Process Model

A program under execution is called a process. The eXpOS associates a virtual (memory) address space for each process. The eXpOS logically partitions the address space into four regions: library, code, stack and heap. These regions are mapped into physical memory using hardware mechanisms like paging/segmentation.

2.3 InterProcess Communication

eXpOS assumes a single processor multi programming environment. It allows processes to communicate with each other using mechanisms like semaphores and WaitSignal system calls. eXpOS provides (binary) semaphores to allow application programs to handle the critical section problem. eXpOS provides system calls like Semget, SemLock, SemUnlock, Semrelease for working with semaphores.

2.4 Shared Memory Model

Shared memory is an efficient means of sharing data between programs. In eXpOS, this sharing is done between the parent process and child processes (or any child process in the hierarchy) through heap. It is the responsibility of the programmer to ensure exclusive access to the shared resources for each process, to avoid data inconsistency. eXpOS helps programmer to realize data consistency with the help of semaphores.

2.5 System Calls

Application programmers interact with the Operating System using the system calls. When a process invokes a system call, the process is interrupted and control goes to the corresponding interrupt service routine of the kernel, resulting in a switch from user mode to kernel mode. Once the system call is carried out, the control goes back to the application program, with a switch back to the user mode.
The following system calls are present in the system:

- File System Calls : Create, Delete, Open, Close, Read, Write, Seek
- Process System Calls : Fork, Exec, Exit, Getpid, Getppid, Shutdown
- System calls for access control and Synchronization: Wait, Signal, FLock, FUnLock, Semget, Semrelease, SemLock, SemUnLock.

2.6 Pre-Emptive Scheduling

In Pre-Emptive Scheduling, process can be paused before its time slice is over. This usually happens when a resource that the process requires is not available at the present. The process puts itself to sleep and another process is scheduled for execution.

2.7 Asynchronous disk operations

To minimize processor cycles spent on disk operations, disk operations are made asynchronous. This means that while a disk operation is being carried out, other processes which do not require the disk can be executed.

3 Design of eXpOS

3.1 Data Structures

The OS data structures store information about processes, files and semaphores. eXpOS data structures can be divided into - Disk Data structures and Memory (in-core) data structures. A copy of Disk Data Structures will be kept in the memory while the system is running.

3.2 Disk Data Structures

3.2.1 Inode Table

Inode Table is the record of the files stored in the disk. The entry of an Inode table has the following format:

FILE TYPE	FILE NAME	FILE SIZE	DATA BLOCK 1	DATA BLOCK 2	DATA BLOCK ...	DATA BLOCK n

Figure 1: Structure of the Inode Table

3.2.2 Disk Free List

For each block in the disk there is an entry in the Disk Free List which contains a value of either 0 or 1 indicating whether the corresponding block in the disk is free or used.

3.3 Memory Data Structures

3.3.1 Process Table

The Process Table contains an entry for each process. Each entry contains several fields that stores all the information pertaining to a single process. The maximum number of entries is equal to maximum number of processes allowed to exist at a single point of time in eXpOS.

TICK	PID	PPID	STATE	MACHINE STATE	PTBR	PTLR	PER-PROCESS RESOURCE TABLE	INODE INDEX	KERNEL STACK POINTER

Figure 2: Structure of the Process Table

The first entry Tick keeps track of how long the process was in memory.

The second column is PID or Process ID which is a number that is unique to each process.

The third column gives the process descriptor of the parent process or the PPID.

Next column, State , consists of a two tuple that describes the current state of the process.

The fifth column is the pointer to a structure that gives the Machine State when the process was last executed. This part is machine dependent.

The next two columns are regarding the page table of the process. The first one (PTBR or Page Table Base Register) stores the starting address of the page table of a process while the next one (PTLR or Page Table Length Register) stores the number of entries in the page table of a process and determines the size of the virtual address space of the process.

The next column is a pointer to a table, the Per-Process Resource Table that contains information about the files opened by the process as well as semaphores used by the process.

Inode Index is a reference to the Inode entry of the executable file. It could be used to access the code pages of the process.

Each process has its own kernel stack. The pointer to the kernel stack is given in the Kernel Stack Pointer column. A process uses its kernel stack to save the return address when it voluntarily schedule out itself inside a blocking system call.

3.3.2 Per-Process Resource Table

This table stores information about the resources (files/semaphores) acquired by the process. For every file opened by the process, it stores the index of the File Table entry for the file, and the LSEEK position of the open instance of a file. The LSEEK position indicates the location in the file where the next read/write operation occurs. Similarly, for every semaphore used by the process, the Per-Process resource table stores the index of Semaphore Table entry.

Index of File Table/ Semaphore Table Entry	LSEEK

Figure 3: Structure of the Per-Process Resource Table

3.3.3 Page Table

The Page Table contains information relating to the actual location of the pages of the process in the memory. The page table contains physical page numbers corresponding to logical pages in the virtual address space of the process. Each entry has a reference bit and a valid bit. Reference bit indicates whether the page is referenced by the process or not. Valid bit indicates whether the page is present in memory or not.

Physical Page Number	Auxiliary Information

Figure 4: Structure of Page Table

3.3.4 File Table

File Table has information about all the files that are currently open. The Open system call creates an entry in the File table when a process opens a file that is not opened by any process in execution. If the file is opened again by some other process (or the same process), the Open system call updates the same file table entry.

Pointer to the Inode entry	Number of Open Instances of the file	PID of the process which has currently locked the file

Figure 5: Structure of the File Table

3.3.5 Semaphore Table

Semaphore Table contains details about all the semaphores used by the processes. For every semaphore opened by a process, there is an entry in the Per-Process resource table, and this entry points to a corresponding entry in the semaphore table.

PID of the process locking the semaphore	Number of processes using the semaphore

Figure 6: Structure of the Semaphore Table

3.3.6 Disk Status Table

eXpOS makes use of Disk Status Table to keep track of load and store operations. It consists of a bit to determine the type of disk operation (load/store), the numbers of page and block involved in the disk operation and the process identifier of the process which initiated disk transfer.

LOAD/STORE BIT	PAGE NUMBER	BLOCK NUMBER	PID of the process who invoked the device

Figure 7: Structure of the Disk Status Table

3.3.7 Buffer Table

To minimize the number of load and store operations, eXpOS provides a buffer cache in memory which would temporarily store disk blocks. Each disk block is mapped to a unique buffer. The disk blocks will be stored back to the disk when some other blocks replace it. Only modified blocks are written back to disk. Buffer Table keeps the information about disk blocks present in the buffer.

Block number of the disk block stored in buffer	DIRTY BIT which indicates whether the block stored in the buffer has been modified	The PID of the process which has currently locked the buffer.

Figure 8: Structure of the Buffer Table

3.3.8 System Status Table

System Status Table keeps the information about number of free pages in memory (MEM_FREE_COUNT), number of processes waiting for memory pages (WAIT_MEM_COUNT), and also the number of processes which have been swapped out to the disk (SWAPPED_COUNT).

3.3.9 Memory Free List

The Memory free list is a data structure used for keeping track of used and unused pages in the memory. Each entry of the free list contains a value of either 0, indicating whether the corresponding page in the memory is free or a number (>0), indicating the number of processes that share the page.

3.4 Algorithms

3.4.1 File System Calls

1. Create System Call

The Create operation takes as input a filename and creates an empty file by that name. If a root entry for the file already exists, then the system call returns 0 (success). Otherwise, it creates a root entry for the file name, sets the file type to DATA and file size to 0. Note that the file name must be a character string and must not be root.

Arguments : Filename

Return Value : 0 (Success) or -1 (No Space for file)

2. Delete System Call

Delete removes the file from the file system and removes its root entry. A file that is currently opened by any application cannot be deleted. Root file also cannot be deleted.

Arguments : Filename

Return Value : 0 (Success) or -1 (File not found) or -2 (File is open)

3. Open System Call

For a process to read/write a file, it must first open the file. Only data and root files can be opened. The Open operation returns a file descriptor. An application can open the same file several times and each time, a different descriptor will be returned by the Open operation. The file descriptor must be passed as argument to other file system calls, to identify the open instance of the file.

The OS associates a file pointer with every open instance of a file. The file pointer indicates the current location of file access (read/write). The Open system call sets the file pointer to 0 (beginning of the file).

Arguments : Filename

Return Value : File Descriptor (Success) or -1 (File not found) or -2 (Process has reached its limit of resources) or -3 (System has reached its limit of open files)

4. Close System Call

After all the operations are done, the user closes the file using the Close system call. The file descriptor ceases to be valid once the close system call is invoked.

Arguments : File Descriptor

Return Value : 0 (Success) or -1 (File Descriptor is invalid)

5. Read System Call

The file descriptor is used to identify an open instance of the file. The Read operation reads one word from the position pointed by the file pointer and stores it into the buffer. After each read operation, the file pointer advances to the next word in the file.

Arguments : File Descriptor and a Buffer (a String/Integer variable) into which a word is to be read from the file

Return Value : 0 (Success) or -1 (File Descriptor is invalid) or -2 (File pointer has reached the end of file)

6. Write System Call

The file descriptor is used to identify an open instance of the file. The Write operation writes the word stored in the buffer to the position pointed by the file pointer of the file. After each Write operation, the file pointer advances to the next word in the file.

Arguments :File Descriptor and a Word to be written

Return Value : 0 (Success) or -1 (File Descriptor given is invalid) or -2 (No disk space)

7. Seek System Call

The Seek operation allows the application program to change the value of the file pointer so that subsequent Read/Write is performed from a new position in the file. The new value of the file pointer is determined by adding the offset to the current value. (A negative Offset will move the pointer backwards). An Offset of 0 will reset the pointer to the beginning of the file.

Arguments : File Descriptor and Offset

Return Value : 0 (Success) or -1 (File Descriptor given is invalid) or -2 (Offset value moves the file pointer to a position outside the file)

3.4.2 Process System Calls

1. Fork System Call
Replicates the process invoking the system call. The heap, code and library regions of the parent are shared by the child. A new stack is allocated to the child and the parent's stack is copied into the child's stack.
When a process executes the Fork system call, the child process shares with the parent all the file and semaphore descriptors previously acquired by the parent. Semaphore/file descriptors acquired subsequent to the fork operation by either the child or the parent will be exclusive to the respective process and will not be shared.

Arguments : None

Return Value : Process Identifier to the parent process and 0 to child process (Success) or -1 (Number of processes has reached its maximum, returned to parent)

3. Exit System Call

Exit system call terminates the execution of the process which invoked it and destroys its memory address space. The calling application ceases to exist after the system call and hence the system call never returns.

Arguments : None

Return Value : -1 (Failure)

4. Getpid System Call
Returns the process identifier of the invoking process. The system call does not fail.

Arguments : None

Return Value : Process Identifier (Success)

5. Getppid System Call
Returns to the calling process the value of the process identifier of its parent. The system call does not fail.

Arguments : None

Return Value : Process Identifier (Success)

6. Shutdown System Call
Shutdown system call terminates all processes and halts the machine.

Arguments : None

Return Value : None

3.4.3 System calls for access control and synchronization

1. Wait System Call
The current process is blocked till the process with PID given as argument executes a Signal system call or exits. Note that the system call will fail if a process attempts to wait for itself.

Arguments : Process Identifier of the process for which the current process has to wait.

Return Value : 0 (Success) or -1 (Given process identifier is invalid or it is the pid of the invoking process)

2. Signal System Call
All processes waiting for the signaling process are resumed. The system call does not fail.

Arguments : None

Return Value : 0 (Success)

4. FUnLock System Call
FUnLock operation allows an application program to unlock a file which the application had locked earlier, so that other applications are no longer restricted from accessing the file.

Arguments : File Descriptor

Return Value : 0 (Success) or -1 (File Descriptor is invalid) or -2 (File was not locked by the calling process)

5. Semget System Call
This system call is used to obtain a binary semaphore. eXpOS has a fixed number of semaphores. The calling process can share the semaphore with its child processes using the fork system call.

Arguments : None

Return Value : semaphore descriptor (Success) or -1 (Process has reached its limit of resources) or -2 (Number of semaphores has reached its maximum)

6. Semrelease System Call
This system call is used to release a semaphore descriptor held by the process.

Arguments : Semaphore Descriptor

Return Value : 0 (Success) or -1 (Semaphore Descriptor is invalid)

7. SemLock System Call
This system call is used to lock the semaphore. If the semaphore is already locked by some other process, then the calling process goes to sleep and wakes up only when the semaphore is unlocked. Otherwise, it locks the semaphore and continues execution.

Arguments : Semaphore Descriptor

Return Value : 0 (Success or the semaphore is already locked by the current process) or -1 (Semaphore Descriptor is invalid)

8. SemUnLock System Call
This system call is used to unlock a semaphore that was previously locked by the calling process. It wakes up all the processes which went to sleep trying to lock the semaphore while the semaphore was locked by the calling process.

Arguments : Semaphore Descriptor

Return Value : 0 (Success) or -1 (Semaphore Descriptor is invalid) or -2 (Semaphore was not locked by the calling process)

3.4.4 Miscellaneous

1. Exception Handler
If a process generates an illegal instruction, an invalid address (outside its virtual address space) or do a division by zero (or other faulty conditions which are machine dependent) or if a page fault occurs, the machine will generate an exception.
The exception handler must terminate the process, wake up all processes waiting for it (or resources locked by it) and invoke the scheduler to continue round robin scheduling the remaining processes.

2. Disk Interrupt Handler
Once a disk transfer is completed, the disk controller produces an interrupt to let the processor know that the disk transfer is done. Disk Interrupt Handler wakes up all processes that went to sleep waiting for the disk while the transfer was going on.

3. Timer Interrupt Handler
The hardware requirement specification for eXpOS assumes that the machine is equipped with a timer device that sends periodic hardware interrupts. The OS scheduler is invoked by the hardware timer interrupt handler. This scheduler uses Round Robin scheduling to schedule the next process for execution.

4 Work Done

- The existing OS data structures were redesigned to incorporate the changes done.
- New data structures like Buffer Table, Disk Status Table, Semaphore Table, System Status Table etc were designed.
- System calls were redesigned to incorporate asynchronous operations, buffer cache and pre-emptive scheduling.
- Directory structure was introduced in eXpFS.
- Executable file format was designed.
- Algorithms for system calls and other interrupt handlers were designed.
- Webpage for Project eXpOS was created. *http://exposnitc.github.io/*

5 Future Work

Future work includes implementation of all the features mentioned above, testing of the system and building a webpage which will act as a reference manual for anyone wishing to build eXpOS.

6 Conclusion

This project aims to create a simpler version of an operating system which allows students to acquire insight into the working of a real operating system.

References

[1] *http://xosnitc.github.io/*

[2] The Design of Unix Operating System, By Maurice J. Bach