Enhancement to eXpOS Operating System and eXpFS File System

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

March 1, 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 (Application Programmer's Language).

1 Problem Definition

This project aims to modify and update the eXpOS operating system by

- Re-engineering the eXpFS system.
- Including inter process communication
- Redesigning process model
- Introducing shared memory model
- Adding more system calls thus broadening the features

Thus, adding the above features, eXpOS would become more advanced and closer to operating systems that are available in the market.

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, Lock, Unlock, 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 three 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 Wait-Signal 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 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

When a process invokes a system call, the process is interrupted and control goes to the kernel. Once the system call is carried out, the control goes back to user mode. There are system calls associated with processes, files and semaphores.
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. It puts itself to sleep and another process can be scheduled for execution.

2.7 Asynchronous disk operations

To minimize processor cycles spent on disk operations, disk operations were 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

It 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 on, for reducing disk access.

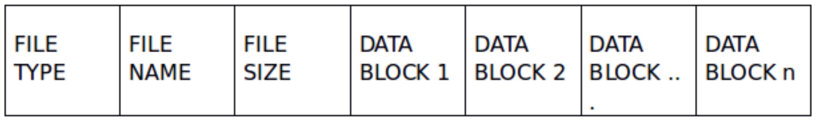
3.2 Disk Data Structures

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:

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, respectively.



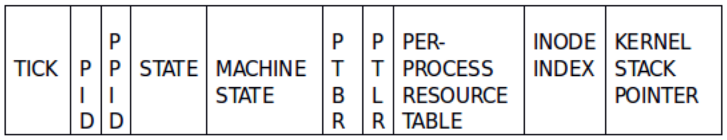
FILE TYPE	FILE NAME	FILE SIZE	DATA BLOCK 1	DATA BLOCK 2	DATA BLOCK ...	DATA BLOCK n

Figure 1: Structure of the Inode Table

3.3 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 and the maximum number of entries is equal to maximum number of processes allowed to exist at a single point of time in eXpOS.



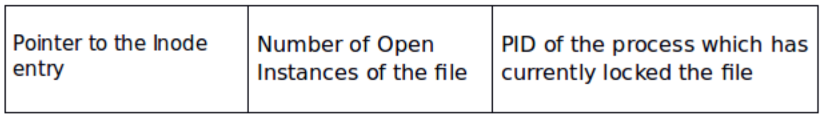
TICK	PID	PPID	STATE	MACHINE STATE	PTBR	PTLR	PER-PROCESS RESOURCE TABLE	INODE INDEX	KERNEL STACK POINTER

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 the unique process descriptor, 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 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.The index of the per-process resource table entry for each resource is known as the resource descriptor.
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 whose pointer is given in the Kernel Stack Pointer column. A process uses its kernel stack to save the return address when it decides to voluntarily schedule out itself out when entering a blocking system call.

3.3.2 File Table

File Table has information about all the files that are currently open. It is initialized by the OS startup code. Initially, there are no open files.

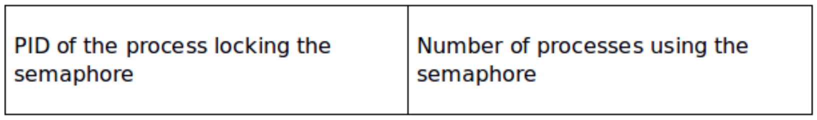


Pointer to the Inode entry	Number of Open Instances of the file	PID of the process which has currently locked the file

Figure 3: Structure of the File Table

3.3.3 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 (which is discussed later), and this entry points to a corresponding entry in the semaphore table.

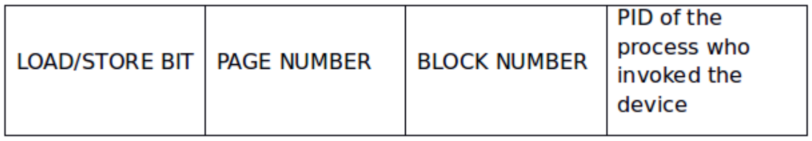


PID of the process locking the semaphore	Number of processes using the semaphore

Figure 4: Structure of the Semaphore Table

3.3.4 Disk Status Table

eXpOS makes use of Disk Status Table to keep track of load and store operations. It consists of four entries which is given in Figure 4.



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

Figure 5: Structure of the Disk Status Table

3.3.5 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 6: Structure of the Buffer Table

3.3.6 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 in swapped state (Swapped_count).

3.3.7 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 number indicating the number of processes that share the page.

3.4 Algorithms

3.4.1 File System Calls

Create System Call
The Create operation takes as input a filename and creates an empty file by that name. It also creates the root entry for the file and makes sure that every file has a unique name. Note that the file name must be a character string and must not be named root.

Arguments : Filename

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

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)

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. The file descriptor conceptually refers to the location of the file which will be referred to by other file operations. The file pointer initially points

Algorithm 1 Create system call

```
if file is present in Inode Table then
  return 0
end if
if no free entry in Inode Table then
  return -1
else
  store index of free entry in InodeIndex
end if
In the Inode Table entry corresponding to InodeIndex, set file name to Filename, file size to 0 and file type to DATA .
In the root file entry corresponding to InodeIndex, set file name to Filename, file size to 0 and file type to DATA .
return 0
```

Algorithm 2 Delete system call

```
if file is not present in Inode table then
  return -1
else
  store index of file in InodeIndex
end if
if file is open then
  return -2
end if
Free all the blocks allocated to the file
Invalidate the Inode Table entry corresponding to Inodeindex
Remove root file entry corresponding to Inodeindex
return 0
```

to the first word in the file when Open system call is completed. An application can open the same file several times and each time a different descriptor will be returned by the Open operation.

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)

Algorithm 3 Open system call

```
if file is not found in Inode Table then
  return -1
end if
if file type is not DATA or ROOT then
  return -1
end if
if no free entry in Per-Process Resource table then
  return -2
else
  store index of free entry in PerProcessIndex
end if
if file is already open then
  store index of file table entry in FTIndex
else
  if no free entry in FT then
    return -2
  else
    store index of free entry in FTIndex
  end if
end if
In entry corresponding to PerProcessIndex, set pointer to FT as FTIndex and LSEEK as 0
In entry corresponding to FTIndex, set pointer to Inode Table as InodeIndex
Increment file open count, set lock status to free
return PerProcessIndex
```

Close System Call
After all the operations are done, the user closes the file using Close system call.

Arguments : File Descriptor

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

Read System Call
Read reads one word into a string/integer variable from the position pointed by file pointer in the file specified by the input file descriptor. After each read is performed, the file pointer advances to the next word in the file.

Algorithm 4 Close system call

```
if File Descriptor is not valid then
  return -1
end if
if entry in Per-Process Resource table is not valid then
  return -1
else
  store the pointer to FIle Table in FTIndex
end if
Decrement file open count
if file is locked by current process then
  unlock the file
end if
if file open count becomes zero then
  Invalidate the File Table entry
end if
Invalidate PerProcess Table entry
return 0
```

Arguments : File Descriptor and a Buffer to which the word is read

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

Write System Call
Write writes one word from a string/integer variable provided by the user, to the position pointed by input file pointer. After each write is performed, 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)

Seek System Call
If user wants to change the file pointer without reading/writing, Seek system call is used. The Seek operation allows the application program to change the location pointed to by the file pointer from the current position to a different position in the file so that next Read/Write is performed from the new position. An Offset of 0 will reset the pointer to the start of the file. A negative Offset will move the pointer backwards.

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

Fork System Call
Replicates the process invoking the system call. The heap, code and

Algorithm 5 Read system call

```
if File Descriptor is not valid then
    return -1
end if
if entry in Per-Process Resource table is not valid then
    return -1
else
    store the pointer to FIle Table in FTIndex
    store LSEEK in lseek
end if
while file is locked do
    put the current process to sleep
    call scheduler
end while
lock the file
store pointer to Inode Table in InodeIndex
if file pointer is at the end of the file then
    return -2
else
    store lseek/XFS_MAXBSIZE in BlockNum
    store lseek%XFS_MAXBSIZE in Offset
end if
find the buffer to which the block is mapped
if buffer is locked then
    if current process has not locked the buffer then
        while buffer is locked do
            put the process to sleep
            call scheduler
        end while
        lock the buffer
    end if
else
    lock the buffer
end if
if buffer does not have required block then
    if buffer contains a block and dirty bit is set then
        store the block in buffer to disk
    end if
    load the required block to the buffer
end if
Read the data
Increment the file pointer
unlock the buffer and wake all processes waiting for the buffer
Unlock the file and wake all processes waiting for the file
return 0
```

Algorithm 6 Write system call

```
if File Descriptor is not valid then
  return -1
end if
if entry in Per-Process Resource table is not valid then
  return -1
else
  store the pointer to FIle Table in FTIndex and LSEEK in lseek
end if
while file is locked do
  put the current process to sleep and call scheduler
end while
lock the file
store pointer to Inode Table in InodeIndex
store lseek/XFS_MAXBSIZE in BlockNum and lseek%XFS_MAXBSIZE in
Offset
if entry corresponding to BlockNum is invalid then
  if no free block in disk then
    return -2
  else
    allocate a free block to the file
    increment file size in Inode Table and root file
  end if
end if
find the buffer to which the block is mapped
if buffer is locked then
  if current process has not locked the buffer then
    while buffer is locked do
      put the process to sleep
      call scheduler
    end while
    lock the buffer
  end if
end if
lock the buffer
if buffer does not have required block then
  if buffer contains a block and dirty bit is set then
    store the block in buffer to disk
  end if
  load the required block to the buffer
end if
Write the data
unlock the buffer and wake all processes waiting for the buffer
Unlock the file and wake all processes waiting for the file
return 0
```

Algorithm 7 Seek system call

```
if File Descriptor is not valid then
  return -1
end if
if entry in Per-Process Resource table is not valid then
  return -1
else
  store the pointer to FIle Table in FTIndex
  store LSEEK in lseek
end if
while file is locked do
  put the current process to sleep
  call scheduler
end while
lock the file
store pointer to Inode Table in InodeIndex
if new file pointer is not valid then
  return -2
end if
if Offset is zero then
  set LSEEK to beginning of file
else
  set LSEEK to LSEEK+Offset
end if
Unlock the file and wake all processes waiting for the file
return 0
```

library regions of the parent are shared by the child. A new stack is allocated to the child and the data in the parent stack is copied into the child stack.
When a process executes the Fork system call, the child process shares with the parent all the file handles and semaphores previously opened by the parent. Note that semaphores and the files handles 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)

Algorithm 8 Fork system call

```
if no free entry in process table then
  return -1
else
  store index of free entry in ChildPID
  store pid of parent process in ParentPID
end if
set the PPID field of child process to ParentPID
count the number of stack pages of parent
while equal number of free pages are not present in memory do
  put the process to sleep
  call scheduler
end while
Allocate free pages to the child
Copy the parent's stack to child's stack
Copy the page table entries of parent, except stack entries, to the page table
of child
Copy the machine state and perprocess resource table to child
For every open file of the parent, increment the file opern count
For every semaphore acquired by the parent, increment process count
set state of child to ready
return 0 to the child process and ChildPID to the parent process
```

Exec System Call
Exec loads the executable program into the memory address space of the calling process, overlaying the calling application with the newly loaded program. After the Fork system call, if either the parent or the child process loads another program into its virtual address space using the exec system call, then the shared heap is detached from that process and the surviving process will have the heap intact. A successful Exec operation results in extinction of the calling application and never returns to the caller application. For the new process, no heap pages are allocated. All open instances of file and semaphores of parent process is closed.

Arguments : Filename

Return Value : -1 (File not found or file is of invalid type)

Algorithm 9 Exec system call

```
if file not found in Inode Table then
  return -1
else
  if file type is not EXEC then
    return -1
  else
    store index of Inode Table entry in *InodeIndex*
    store the ode block numbers of the file in *Block1* and *Block2*.
  end if
end if
In the page table of current process, set code page entries to *Block1* and *Block2*.
Include the page numbers of shared library in the page table.
Set the auxillary information to invalid and unreferenced.
Invalidate the entry for heap pages
Close all files opened by the current process
Release all semaphores held by the parent process.
Set SP and IP values to valid locations.
return 0
```

Exit System Call
Exit system call terminates the execution of the process which invoked it and removes it from the memory. It schedules the next ready process and starts executing it. When there is no other ready process to run, it halts the machine.

Arguments : None

Return Value : -1 (Failure)

Getpid System Call
Returns the process identifier of the invoking process.

Arguments : None

Return Value : Process Identifier (Success) or -1 (Failure)

Getppid System Call
Returns the process Identifier of the parent of the invoking process.

Arguments : None

Return Value : Process Identifier (Success) or -1 (Failure)

Shutdown System Call Shutdown system call terminates all processes, writes modified(dirty) pages and memory copy of disk data structures to the disk, and then halts the machine. In case the system call failed, an error is returned to the invoking process.

Algorithm 10 Exit system call

```
if no more processes to schedule then
    shutdown the machine
else
    store the pid of the next ready process in NextPID
end if
Close all files opened by the current process
Release all the semaphores used by the current process
Memory pages of the current process are freed
Invalidate the page table entry
Wake up all processes waiting for the current process
Schedule the process with pid NextPID
return
```

Algorithm 11 Getpid system call

```
Find the PID of the current process by using PTBR value.
return  PID of current process
```

Arguments : None

Return Value : -1 (failure)

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 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.

5 Future Work

Future work includes implementation of all the features mentioned above, testing of the system and building a roadmap for anyone wishing to build eXpOS.

Algorithm 12 Getppid system call

```
Find the PID of the current process by using PTBR value.
From the Process Table entry of the current process, find the PPID
return  PPID of current process
```

Algorithm 13 Shutdown system call

```
while disk is not free do
    put the process to sleep
    call scheduler
end while
store Inode Table to the disk
store dirty pages to disk
store Disk Free List to the disk
Halt the machine
return
```

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